JSC Guest Student Programme Proceedings 2016

Edited by Ivo Kabadshow

JÜLICH
FORSCHUNGSZENTRUM


data
model
matrix
size
approach
rank
value
system
program
performance
thread
intel
array
gpu
method
based
memory
left
scale
limit
parallel
bandwidth
task
pixel
mpi
function
grid
set
brain
threads
particles
element
algorithm
called
gray
space
shown
cuda
filter
result
time
cpu
hand
floor
step
due
int
field
cell
section
implementation
local
values
communication
node
zero
operations
lattice
boundary
code
version
processes
layer
cells
list
particle
single
buffer
http
x1
x2
x3
x4
x5
x6
x7
x8
x9
x10
x11
x12
x13
x14
x15
x16
x17
x18
x19
x20
x21
x22
x23
x24
x25
x26
x27
x28
x29
x30
x31
x32
x33
x34
x35
x36
x37
x38
x39
x40
x41
x42
x43
x44
x45
x46
x47
x48
x49
x50
x51
x52
x53
x54
x55
x56
x57
x58
x59
x60
x61
x62
x63
x64
x65
x66
x67
x68
x69
x70
x71
x72
x73
x74
x75
x76
x77
x78
x79
x80
x81
x82
x83
x84
x85
x86
x87
x88
x89
x90
x91
x92
x93
x94
x95
x96
x97
x98
x99
x100
x101
x102
x103
x104
x105
x106
x107
x108
x109
x110
x111
x112
x113
x114
x115
x116
x117
x118
x119
x120
x121
x122
x123
x124
x125
x126
x127
x128
x129
x130
x131
x132
x133
x134
x135
x136
x137
x138
x139
x140
x141
x142
x143
x144
x145
x146
x147
x148
x149
x150
x151
x152
x153
x154
x155
x156
x157
x158
x159
x160
x161
x162
x163
x164
x165
x166
x167
x168
x169
x170
x171
x172
x173
x174
x175
x176
x177
x178
x179
x180
x181
x182
x183
x184
x185
x186
x187
x188
x189
x190
x191
x192
x193
x194
x195
x196
x197
x198
x199
x200
x201
x202
x203
x204
x205
x206
x207
x208
x209
x210
x211
x212
x213
x214
x215
x216
x217
x218
x219
x220
x221
x222
x223
x224
x225
x226
x227
x228
x229
x230
x231
x232
x233
x234
x235
x236
x237
x238
x239
x240
x241
x242
x243
x244
x245
x246
x247
x248
x249
x250
x251
x252
x253
x254
x255
x256
x257
x258
x259
x260
x261
x262
x263
x264
x265
x266
x267
x268
x269
x270
x271
x272
x273
x274
x275
x276
x277
x278
x279
x280
x281
x282
x283
x284
x285
x286
x287
x288
x289
x290
x291
x292
x293
x294
x295
x296
x297
x298
x299
x300
x301
x302
x303
x304
x305
x306
x307
x308
x309
x310
x311
x312
x313
x314
x315
x316
x317
x318
x319
x320
x321
x322
x323
x324
x325
x326
x327
x328
x329
x330
x331
x332
x333
x334
x335
x336
x337
x338
x339
x340
x341
x342
x343
x344
x345
x346
x347
x348
x349
x350
x351
x352
x353
x354
x355
x356
x357
x358
x359
x360
x361
x362
x363
x364
x365
x366
x367
x368
x369
x370
x371
x372
x373
x374
x375
x376
x377
x378
x379
x380
x381
x382
x383
x384
x385
x386
x387
x388
x389
x390
x391
x392
x393
x394
x395
x396
x397
x398
x399
x400
x401
x402
x403
x404
x405
x406
x407
x408
x409
x410
x411
x412
x413
x414
x415
x416
x417
x418
x419
x420
x421
x422
x423
x424
x425
x426
x427
x428
x429
x430
x431
x432
x433
x434
x435
x436
x437
x438
x439
x440
x441
x442
x443
x444
x445
x446
x447
x448
x449
x450
x451
x452
x453
x454
x455
x456
x457
x458
x459
x460
x461
x462
x463
x464
x465
x466
x467
x468
x469
x470
x471
x472
x473
x474
x475
x476
x477
x478
x479
x480
x481
x482
x483
x484
x485
x486
x487
x488
x489
x490
x491
x492
x493
x494
x495
x496
x497
x498
x499
x500
x501
x502
x503
x504
x505
x506
x507
x508
x509
x510
x511
x512
x513
x514
x515
x516
x517
x518
x519
x520
x521
x522
x523
x524
x525
x526
x527
x528
x529
x530
x531
x532
x533
x534
x535
x536
x537
x538
x539
x540
x541
x542
x543
x544
x545
x546
x547
x548
x549
x550
x551
x552
x553
x554
x555
x556
x557
x558
x559
x560
x561
x562
x563
x564
x565
x566
x567
x568
x569
x570
x571
x572
x573
x574
x575
x576
x577
x578
x579
x580
x581
x582
x583
x584
x585
x586
x587
x588
x589
x590
x591
x592
x593
x594
x595
x596
x597
x598
x599
x600
x601
x602
x603
x604
x605
x606
x607
x608
x609
x610
x611
x612
x613
x614
x615
x616
x617
x618
x619
x620
x621
x622
x623
x624
x625
x626
x627
x628
x629
x630
x631
x632
x633
x634
x635
x636
x637
x638
x639
x640
x641
x642
x643
x644
x645
x646
x647
x648
x649
x650
x651
x652
x653
x654
x655
x656
x657
x658
x659
x660
x661
x662
x663
x664
x665
x666
x667
x668
x669
x670
x671
x672
x673
x674
x675
x676
x677
x678
x679
x680
x681
x682
x683
x684
x685
x686
x687
x688
x689
x690
x691
x692
x693
x694
x695
x696
x697
x698
x699
x700
x701
x702
x703
x704
x705
x706
x707
x708
x709
x710
x711
x712
x713
x714
x715
x716
x717
x718
x719
x720
x721
x722
x723
x724
x725
x726
x727
x728
x729
x730
x731
x732
x733
x734
x735
x736
x737
x738
x739
x740
x741
x742
x743
x744
x745
x746
x747
x748
x749
x750
x751
x752
x753
x754
x755
x756
x757
x758
x759
x760
x761
x762
x763
x764
x765
x766
x767
x768
x769
x770
x771
x772
x773
x774
x775
x776
x777
x778
x779
x780
x781
x782
x783
x784
x785
x786
x787
x788
x789
x790
x791
x792
x793
x794
x795
x796
x797
x798
x799
x800
x801
x802
x803
x804
x805
x806
x807
x808
x809
x810
x811
x812
x813
x814
x815
x816
x817
x818
x819
x820
x821
x822
x823
x824
x825
x826
x827
x828
x829
x830
x831
x832
x833
x834
x835
x836
x837
x838
x839
x840
x841
x842
x843
x844
x845
x846
x847
x848
x849
x850
x851
x852
x853
x854
x855
x856
x857
x858
x859
x860
x861
x862
x863
x864
x865
x866
x867
x868
x869
x870
x871
x872
x873
x874
x875
x876
x877
x878
x879
x880
x881
x882
x883
x884
x885
x886
x887
x888
x889
x890
x891
x892
x893
x894
x895
x896
x897
x898
x899
x900
x901
x902
x903
x904
x905
x906
x907
x908
x909
x910
x911
x912
x913
x914
x915
x916
x917
x918
x919
x920
x921
x922
x923
x924
x925
x926
x927
x928
x929
x930
x931
x932
x933
x934
x935
x936
x937
x938
x939
x940
x941
x942
x943
x944
x945
x946
x947
x948
x949
x950
x951
x952
x953
x954
x955
x956
x957
x958
x959
x960
x961
x962
x963
x964
x965
x966
x967
x968
x969
x970
x971
x972
x973
x974
x975
x976
x977
x978
x979
x980
x981
x982
x983
x984
x985
x986
x987
x988
x989
x990
x991
x992
x993
x994
x995
x996
x997
x998
x999
x1000

EDITORIAL

High-performance computing and simulation are strongly advancing fields in the scientific community. The Jülich Supercomputing Centre (JSC) provides a high-class HPC infrastructure and fosters young scientists willing to enter these domains. Our ten-week guest student program offers interested students the opportunity to work within one of the world's most powerful HPC environments. Within this programme, students with a major in natural sciences, engineering, computer science or mathematics get the opportunity to familiarize themselves with different aspects of scientific computing. Together with local scientists, the participants work on different topics in research and development. Depending on previous knowledge and on the participant's interest, the assignment can be chosen out of different areas. These fields include mathematics, physics, chemistry, neuroscience, software development tools, visualization, distributed computing, operating systems and communication.

The JSC Guest Student Programme has already been successfully running for 17 years. Since the first programme in 2000, a total of 182 students have seized the opportunity to join research teams from JSC on the Forschungszentrum Jülich campus each summer. Working on challenging scientific projects, they gained experience with modern hardware and software as well as HPC-related methods and algorithms. For many students, the programme has been the foundation for a career in HPC and the basis for fruitful continuing cooperations.

The JSC Guest Student Programme 2016 took place from August 1st to October 7th. Once again it was run under the CECAM framework (Centre Européen de Calcul Atomique et Moléculaire) with support by IBM Deutschland through a sponsorship within the IBM university relations programme. It targeted students who have already completed their first degree but have not yet finished their master's course.

This year's announcement yielded a record response of about 100 applications from over 30 different countries. Competition for the available places was especially strong, and after the final selection process, 13 students were invited to Jülich. This publication summarized the findings of these research projects.

Ivo Kabadshow
Jülich, December 2016

Contents

Photons with Möbius Boundary

Quenched Schwinger model on a Klein bottle

Giovanni Iannelli
Physics Department
University of Pisa
Italy
iannelli.giovanni@gmail.com

Abstract *In this report I will present an implementation of a pure gauge 2D $U(1)$ theory on a lattice. The objective is to measure the topological susceptibility, which, in the full QCD theory, is proportional to the mass squared of the axion, a candidate for dark matter. I will show that a Möbius boundary is useful to avoid charge freezing, the main problem that prevents performing a well controlled extrapolation to the continuum limit. Furthermore, I will discuss how the topological charge is influenced by space topology, and why a new topological invariant is arising.*

1 Introduction

One of the main research topics in modern Physics is the origin and behaviour of Dark Matter.

We know, from observations, that Dark Matter has a mass and generates a gravitational field. However, its peculiarity lies in its apparent complete lack of interaction with all other forces in nature. Its contribution to the total amount of matter (or energy) in the universe is remarkable: it is approximately 26.8 %, while the contribution of ordinary matter is 4.9 % and 68.3 % is Dark Energy [1].

Axion mass

The axion is a hypothetical particle, and it was introduced in 1977 by Peccei and Quinn to solve a problem in CP symmetry of QCD [5]. However, they didn't realize that their model implies the existence of a new lightweight boson, and this was pointed out the following year by Weinberg [7] and Wilczek [8]. They noticed that such a particle must have a very small coupling with the Standard Model, and that's a big clue for being Dark Matter.

In this model, the axion appears coupled to its counterpart, the saxion, and both are described by the complex scalar theory:

$$\mathcal{L} = \partial_\mu \phi^* \partial^\mu \phi - \frac{\lambda}{8} \left(\phi^* \phi - f_a^2\right)^2 + \chi_t \frac{|\phi|}{f_a} \cos \arg \phi$$

in which:

- $\arg \phi$ is the axion field
- $|\phi|$ is the saxion field
- χ_t is the effective coupling with QCD
- f_a is a constant that has units of a mass, and it must be very big ($\sim 10^{10}$ GeV)

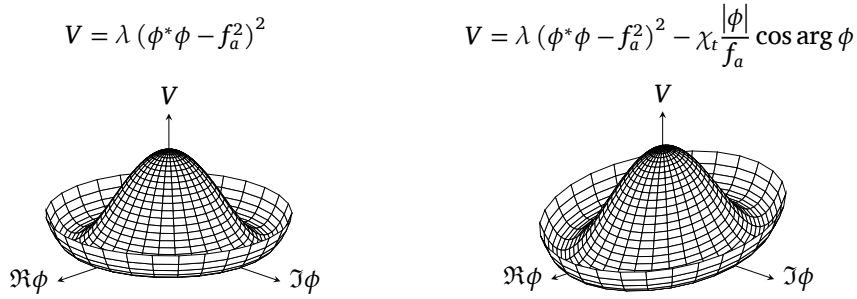


Figure 1.1: Mexican hat potential – effective potential in the axion Lagrangian.

The addition of the coupling term to this Lagrangian creates a new massive mode in the potential, and that is where the axion mass comes from. This process can be seen with the aid of the Mexican hat potential plot in Fig. 1.1: particles are oscillation modes among the minima of the potential V, and, without the coupling term, V has a nonzero slope only in the radial direction. This oscillation mode corresponds to the saxion. If the coupling term is added, there is a slope also along the angular direction that gives rise to the axion mass.

In terms of the Lagrangian parameters, the two masses are:

- Saxion: $m_s = \sqrt{\lambda} f_a$, which is very big due to f_a
- Axion: $m_a = \dfrac{\sqrt{\chi_t}}{f_a}$, which is instead very small because of f_a suppression

Topological susceptibility

The interesting part of this model is that the axion mass is related to a functional Q of the gluon field strength tensor, called topological charge:

$$Q = \frac{1}{4\pi^2} \int \mathrm{d}^4x\, \epsilon_{\mu\nu\rho\sigma} F_{\mu\nu} F_{\rho\sigma}$$

In particular, the topological susceptibility $\dfrac{\langle Q^2 \rangle}{V}$ acquires an important role since it is proportional to the axion mass:

$$\frac{\langle Q^2 \rangle}{V} = \chi_t = f_a^2 m_a^2$$

where $\langle . \rangle$ indicates a mean value and V the volume of the space-time.

This susceptibility can be evaluated through numerical simulations, collecting gluon field configurations and performing the mean value.

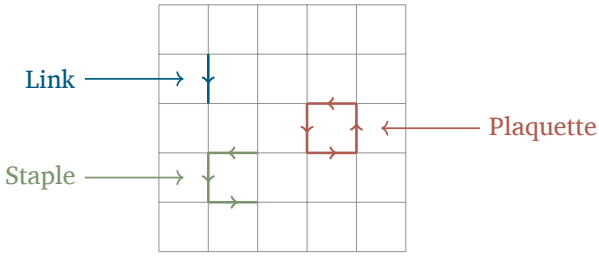


Figure 2.1: Useful objects in the discretized space-time.

2 Schwinger model

When new ideas or algorithms are going to be implemented, it is comfortable to apply them to a simplified model in order to observe the consequences without having to handle a cumbersome theory like QCD.

Instead of the full 4D SU(3) QCD, it's possible to work with the Schwinger model, a 2D U(1) QED theory that shares important properties with QCD, like the confinement of fermions [6].

However, in my simulations, I implemented the model in the quenched approximation, in which the Lagrangian has only the pure gauge term:

$$\mathcal{L} = \frac{1}{4} F_{\mu\nu} F_{\mu\nu}$$

In this case $F_{\mu\nu}$ is the electro-magnetic tensor, and the oscillation modes of the electro-magnetic fields are photons.

Schwinger model on a lattice

To perform numerical simulations, the space-time has to be discretized on a grid, or lattice (Fig. 2.1). Link variables are the lines connecting two lattice sites and are U(1) elements, represented as $\left\{e^{i\varphi}\right\}_{\varphi\in(-\pi,\pi]}$ numbers. If a link is taken in the opposite direction, its value should be the complex conjugate. Plaquettes are given by the multiplication of links in small closed loops and their value is related to the electromagnetic field.

Links have to be sampled from a probability density function e^{-S}, in which

$$S = \beta \sum_{\square} (1 - \Re \square)$$

is the action of the theory, $\square$s are the plaquettes, and β is a constant parameter related to the physical coupling.

Heat-bath algorithm

The link sampling is done through a Markov Chain Monte Carlo: at every step only one link is sampled, leaving the other links unchanged. Q is evaluated after each sweep, which consists in sampling, one by one, every link in the lattice.

To get one link from his distribution, an accept/reject Metropolis-Hastings step is implemented. The procedure I chose to apply is similar to the one showed in Montvay-Münster [3] for extracting SU(2) matrices, but instead of starting from an exponential proposal, I started from a gaussian proposal:

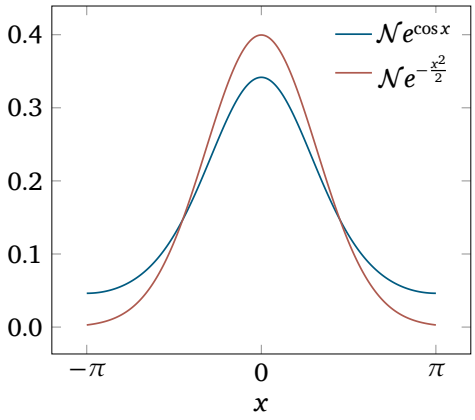


Figure 2.2: The $\mathcal{N}$s are the normalization factors.

Let U be the link to sample, $\hat{U}$ all the other links (kept fixed), S the staples connected to U and $W(U|\hat{U})$ be the probability density function of U. Then

$$W(U|\hat{U}) \propto e^{\beta\Re(US)}$$

Defining $k \equiv |S|, U_0 \equiv \frac{S}{k}U$, it is still $U_0 \in U(1)$, so:

$$W(U|\hat{U})\mathrm{d}U = W(U_0|\hat{U}_0)\mathrm{d}U_0 = e^{\beta k\Re(U_0)}\mathrm{d}U_0 = e^{\beta k\cos\arg U_0}\mathrm{d}U_0$$

since $\mathrm{d}U$ and $\mathrm{d}U_0$ are the same measure.

Due to the fact that $e^{\cos x}$ is similar to the gaussian $e^{1-\frac{x^2}{2}}$ (see fig. 2.2), the accept/reject algorithm is straightforward and the acceptance ratio is pretty high: in all simulations I did, is above 95 %.

Topological charge in the Schwinger model

The topological charge in 2D assumes a simplified form:

$$Q = \frac{1}{4\pi}\int \mathrm{d}^2x\,\epsilon_{\mu\nu}F_{\mu\nu}$$

while, in the discretized theory, it becomes:

$$Q = \frac{1}{2\pi}\sum\nolimits_{\square}(\arg\square \to (-\pi,\pi])$$

in which $\arg\square \to (-\pi,\pi]$ means that the angle given by $\arg\square$ is shifted to the equivalent value in $(-\pi,\pi]$, and this is done by the function $x \mapsto x - \left\lceil\frac{x-\pi}{2\pi}\right\rceil 2\pi$.

The properties of this functional can help to visualize the physical meaning of Q:

- Different areas, called instantons, with their own charge value, contribute to the total amount of Q.

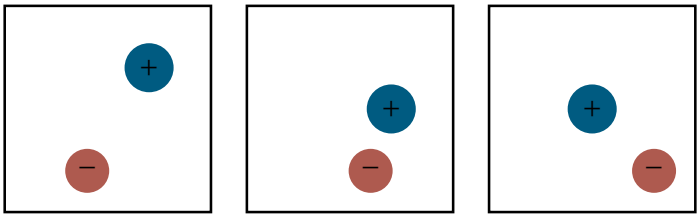


Figure 2.3: Instantons moving around in the bulk.

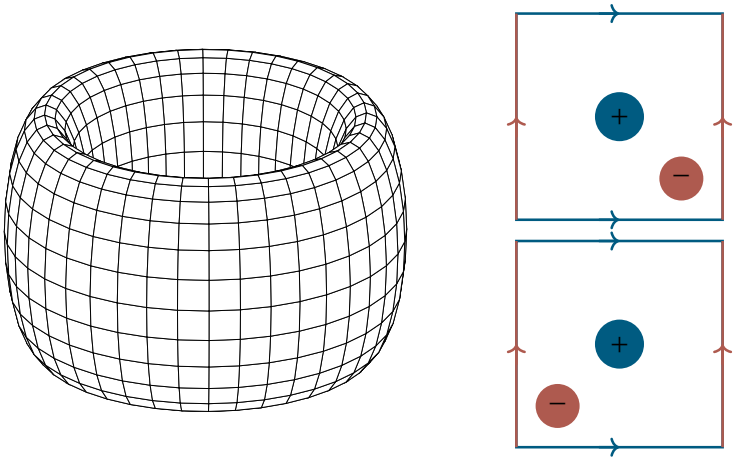


Figure 2.4: Instanton moving over the boundary on a torus.

- If very few local changes are applied to links, inside or not the instantons, the charge value remains constant, and, for this reason, is difficult to create or destroy a new instanton, especially for fine lattices.

- What actually happens, is that instantons just move around in the space during the simulation (Fig. 2.3).

Boundary conditions

Usually periodic boundary conditions are implemented in lattice simulations. This means that the borders of the lattice are connected in order to form a torus, like in Fig. 2.4. In such a case, Q can only assume integer values because, over the sum, every link is taken in both directions, and the only terms surviving are the shifting terms to $(-\pi, \pi]$, which are just integer numbers.

If an instanton reaches the border, it will just be moved on the other side, and the total amount of charge remains constant. This property is called topological invariance. This means that, in the continuum limit, Q will be frozen in only one configuration, and it will be impossible to compute any reliable mean value (Fig. 2.5).

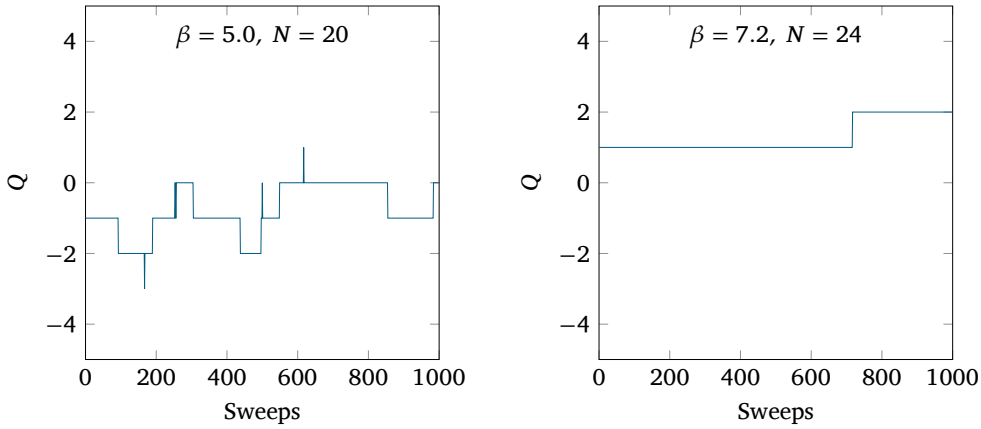


Figure 2.5: History plot of the topological charge on a torus. When the lattice gets finer, the charge freezes in one value. (Periodic boundary).

Möbius boundary

The situation could change and the problem could be solved, if one of the boundaries is switched from periodic to Möbius, as discussed in [4]. Gluing together such boundaries, in 2D, would produce a Klein bottle surface (Fig. 2.6) and Q can now assume non-integer values, because the contribution of links on the Möbius border is not cancelled anymore over the summation.

Another crucial property of the topological charge is that if the orientation of the manifold is inverted, the value of Q changes sign. This means that, on a non-orientable surface, an instanton can change sign just moving through the boundary.

The history plot with Möbius conditions has a different behaviour (Fig. 2.7) and doesn't manifest freezing anymore.

It is interesting to notice that the sign flipping of the instantons can be seen from the history plot: Q has rapid jumps of ± 2 or ± 4, and this means that one or two instantons have passed the Möbius border.

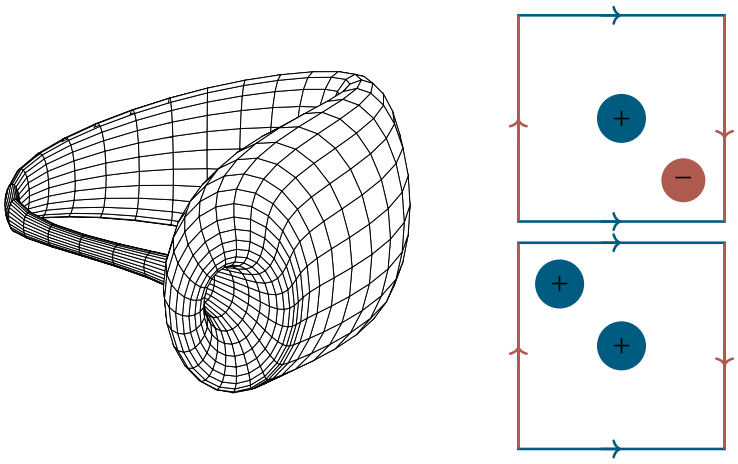


Figure 2.6: Instanton moving over the boundary on a Klein bottle.

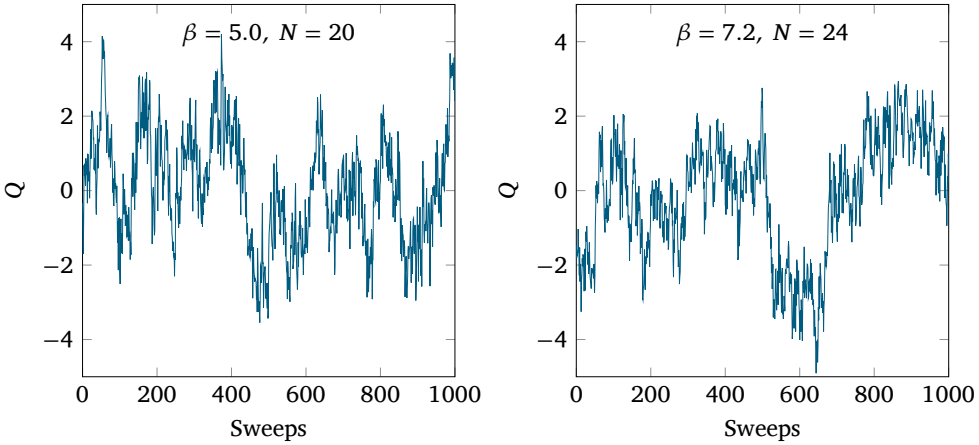


Figure 2.7: History plot of the topological charge on a Klein bottle. The charge doesn't freeze as in the torus case. (Möbius boundary).

3 Numerical results

For all my simulations, I have started with a hot initial configuration, i.e. every link angle is sampled from an uniform distribution in $(-\pi, \pi]$. In each simulation, I made 50000 measures, one every sweep, and discarded the very first thermalization values (I chose the number of iterations to discard accordingly to the corresponding history plot of the charge). To get rid of the autocorrelation bias in evaluating mean values and errors, I applied Jackknife method on the dataset splitted in 20 blocks (20 are enough to have correct Jackknife evaluators, but not too many to show autocorrelation).

Measuring the plaquette

The first thing one would like to measure is the plaquette mean value: it is the best and fastest validity test, since its value is available for comparison in the literature. After each sweep, I evaluated and stored the mean value of all plaquettes in the lattice.

Furthermore, the plaquette mean value in the infinite box size limit should be the same also with Möbius boundary, since it is independent of the orientation of the space.

Running the program with $\beta = 7.2$ and box size $N = 24$, I could check the compatibility between my values the one obtained by Dürr-Hoelbling (2005) [2]:

- 0.927681(78) on a torus
- 0.927729(37) on a Klein bottle
- 0.927722(54) Dürr-Hoelbling

Topological susceptibility: infinite volume limit

To figure out the limit of infinite volume, one should keep the coupling fixed (i.e. β) and compute χ for different values of box size N.

	A	B	C	D	E
β	1.0	1.0	1.0	1.0	1.0
N	12	16	20	24	28

As shown in Fig. 3.1, topological susceptibility is already stable for small box sizes, and the results are compatible with both boundary conditions.

Continuum limit

To evaluate the limit of vanishing lattice, β has to be increased together with the size of the box in order to keep the physical coupling constant. To find the corresponding β values, one needs a scaling study, and I have used the results obtained by Dürr-Hoelbling (2005) [2]:

	F	G	H	I	J
β	1.8	3.2	5.0	7.2	9.8
N	12	16	20	24	28

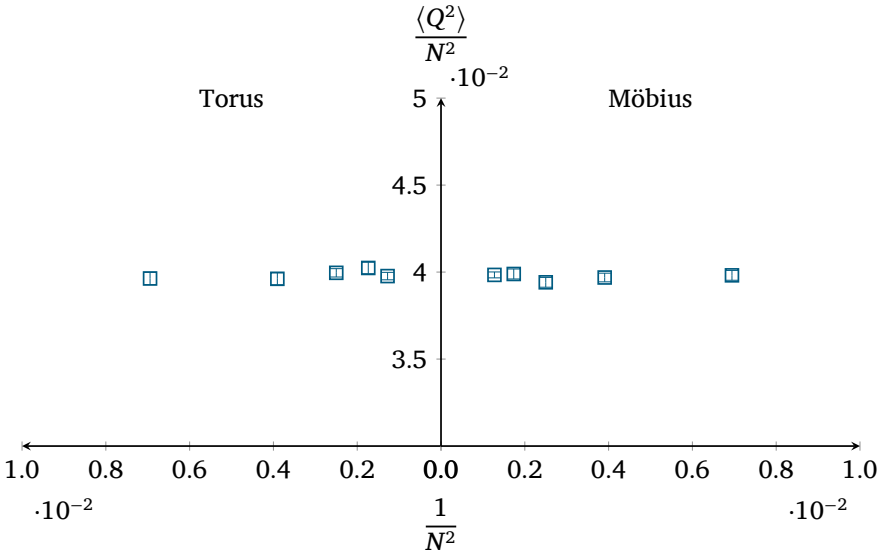


Figure 3.1: Infinite volume limit of the topological susceptibility. In both cases, lattice artifacts are not observed even for small lattices, like 12×12.

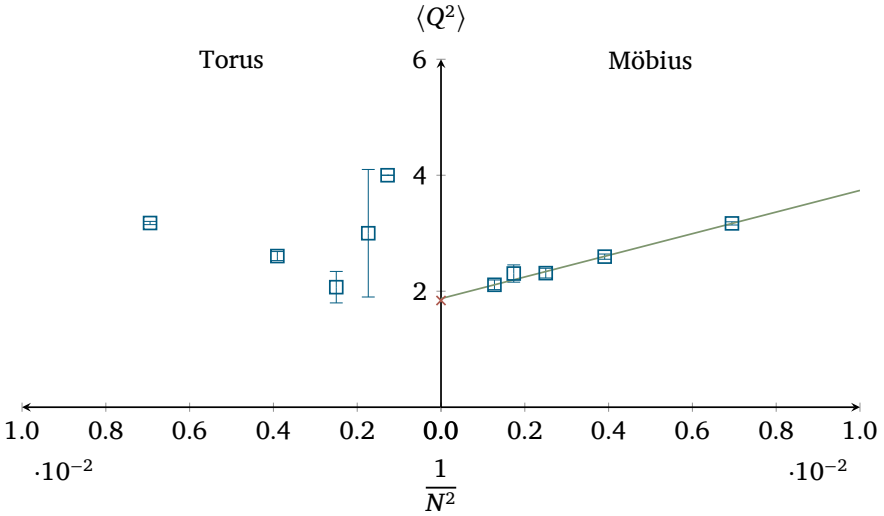


Figure 3.2: Continuum limit of the topological susceptibility. On a torus, data are compromised by charge freezing.

As shown in Fig. 3.2, the values for $\langle Q^2 \rangle$ have the same behaviour under both boundary conditions, however, in the case of periodic boundary, topological freezing prevents a fair evaluation: for point I, the number of independent configurations is not enough to get a useful value, while, for point J, there is complete freezing at the value $Q = 2$. On the other hand, for Möbius boundary, it's possible to perform measurements towards the continuum limit without suffering from charge freezing. In the plot is also shown a linear fit of the data in green, and the red cross indicates the limit value found by Dürr-Hoelbling (2005) [2]:

- $\langle Q^2 \rangle = 1.87(6)$ is the continuum limit on a Klein bottle
- $\langle Q^2 \rangle = 1.84(6)$ is the limit from Dürr-Hoelbling on a torus

4 Conclusion

Changing the boundary condition actually prevents charge freezing and yields the same continuum limit, confirming that this approach could lead to a more precise value of the axion mass.

In the Schwinger model, finite box size artifacts to the susceptibility are very small, even for quite small box sizes, and also switching one boundary to Möbius doesn't produce any significant artifact.

Few other considerations have to be done about the topological charge because both charge quantization and topological invariance are lost switching from a torus to a Klein bottle.

A new charge

In order to recover them, a new topological charge can be defined. To have only integer values, the contribution of the Möbius border have to be subtracted. This is done adding two times the Polyakov loop on the Möbius border. This operation, however, leaves a gauge degree of freedom of $4k\pi$:

$$Q = \frac{1}{2\pi} \sum (\arg \square \to (-\pi, \pi] + 2P_0 + 4k\pi)$$

and can be suppressed taking the modulus 2 value:

$$Q_{e/o} \equiv Q \mod 2$$

This charge is also a topological invariant because, if an instanton changes sign, the variation of Q is absorbed into the modulus operation.

Plotting the history of $Q_{e/o}$ (Fig. 4.1), shows again the charge freezing in one of the two possible values.

Furthermore, for all values of (β, N) considered, the ratio $\dfrac{Q_{e/o} = 1}{Q_{e/o} = 0}$ is compatible with 1. This situation may change if also fermions are considered in the action, and implementing them could be an interesting outlook for this project.

It's not clear what could be the physical meaning of this new charge. Perhaps it could be coupled to the axion mass as well, in an extension of the model that take into consideration also topologically non-trivial manifolds.

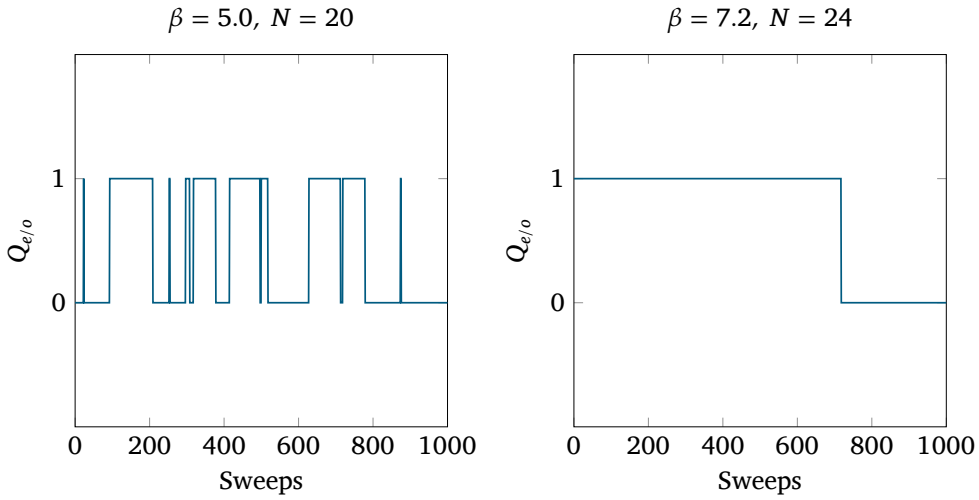


Figure 4.1: History plot of the even/odd charge. Freezing is now present since this charge is a topological invariant

5 Acknowledgements

There are many people I would like to thank, but the first one is certainly Dr. Ivo Kabadshow, who gave me the opportunity to join this programme, and stayed with us for days and nights to help solving our problems.

Then, I want to express my gratitude to my advisor, Prof. Kálmán Szabó, that chose to assign me this beautiful project, and helped me, step by step, to understand the topic and analyze the results. He let me dive into this interesting subject, in which I would like to continue my studies.

At the same time I'm very thankful to his postdoc, Dr. Simon Mages, that came up with the idea of the Möbius boundary, which is the essence of my project. He was also very clear in explanations and very helpful during the debugging process. It was a pleasure to work with him because he was always pleasant and ironic during those times.

In the end, I would like to say the most important thank to my colleagues, that made this period to be special for me. I will miss a lot all the time spent together, and all our crazy but wonderful weekend trips.

References

[1] P. Ade, N. Aghanim, C. Armitage-Caplan, M. Arnaud, M. Ashdown, F. Atrio-Barandela, J. Aumont, C. Baccigalupi, A. J. Banday, R. Barreiro, et al. **Planck 2013 results. XVI. Cosmological parameters**. *Astronomy & Astrophysics*, 571:A16, 2014.

[2] S. Dürr and C. Hoelbling. **Scaling tests with dynamical overlap and rooted staggered fermions**. *Phys. Rev. D*, 71:054501, Mar 2005. http://link.aps.org/doi/10.1103/PhysRevD.71.054501, doi:10.1103/PhysRevD.71.054501.

[3] I. Montvay and G. Munster. *Quantum fields on a lattice*. Cambridge University Press, 1997. http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=0521404320.

[4] S. Mages, B. C. Toth, S. Borsanyi, Z. Fodor, S. Katz, and K. K. Szabo. **Lattice QCD on Non-Orientable Manifolds**. 2015. arXiv:1512.06804.

[5] R. D. Peccei and H. R. Quinn. CP **Conservation in the Presence of Pseudoparticles**. *Phys. Rev. Lett.*, 38:1440–1443, Jun 1977. http://link.aps.org/doi/10.1103/PhysRevLett.38.1440, doi:10.1103/PhysRevLett.38.1440.

[6] J. Schwinger. **Gauge Invariance and Mass. II**. *Phys. Rev.*, 128:2425–2429, Dec 1962. http://link.aps.org/doi/10.1103/PhysRev.128.2425, doi:10.1103/PhysRev.128.2425.

[7] S. Weinberg. **A New Light Boson?** *Phys. Rev. Lett.*, 40:223–226, Jan 1978. http://link.aps.org/doi/10.1103/PhysRevLett.40.223, doi:10.1103/PhysRevLett.40.223.

[8] F. Wilczek. **Problem of Strong *P* and *T* Invariance in the Presence of Instantons**. *Phys. Rev. Lett.*, 40:279–282, Jan 1978. http://link.aps.org/doi/10.1103/PhysRevLett.40.279, doi:10.1103/PhysRevLett.40.279.

FLOOR FIELDS IN JUPEDSIM

Fabian Mack
Faculty of Chemistry and Biosciences
Karlsruhe Institute of Technology
Germany
fabian.mack@student.kit.edu

Abstract *For the JPScore module of JuPedSim, several changes with the aim of speeding up the router and the direction strategy computation when floor fields are used have been made. The implementation ideas are presented in detail in this report. Furthermore, the scaling behavior of the now-improved program has been investigated.*

1 INTRODUCTION

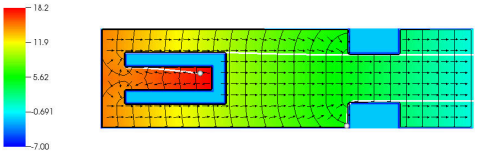
Pedestrian dynamics is a research field of great interest. One branch deals with the calculation of the movement of pedestrian in a given environment and under given circumstances.

Places with high pedestrian density like shopping malls or train stations – buildings which are erected for decades – need to be planned carefully to avoid congestions or long waiting times. Being able to predict the paths pedestrians will take can help architects to optimize the pedestrian traffic flow, allowing them to use the building more economically and more comfortably.

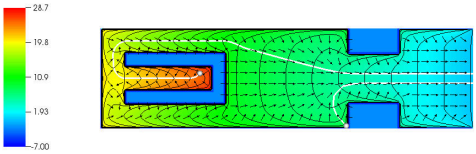
For all these kinds of buildings, as well as for big events (e.g. festivals), it is also necessary to plan escape routes. It is crucial to know the capacity of the location, as well as the evacuation time and the crowd movement in an emergency situation. When it becomes apparent in advance that this causes problems, pedestrian simulation can be used to optimize such route by relocating barriers or adding information signs.

The calculation of the pedestrian movement has to be included into a larger context. An input geometry of the location to be investigated has to be provided, then the simulation can be run. Data has to be collected and statistical analysis has to be performed in order to use the generated information or to validate the model. It might also be desirable to visualize the trajectories of the pedestrians to easily identify bottlenecks. One framework which provides this functionality is the Jülich Pedestrian Simulator (JuPedSim), a platform independent open-source project, licensed under the GNU Lesser General Public License (LGPL) [3]. The aforementioned tasks are mainly independent from each other, therefore, JuPedSim is divided into several modules.

The calculation of the trajectories is done in a module named JPScore. There are different tasks to be done for a complete simulation. First, the pedestrians need their next destination (which usually is a door). It is the *router*'s task to find the optimal route and guide the pedestrians through the building. Second, the pedestrian needs to know how to navigate from his position inside the room to the door designated by the router. The direction a pedestrian should go in is given by the so called *direction strategy*. Third, a pedestrian cannot always walk in this desired direction: Other pedestrians might be standing in his way. The *operational model* calculates the interaction between the pedestrians and prevents them from walking into each other. This is done in parallel for all the pedestrians, one time step after the other. This is necessary because the latest position of all pedestrians is needed for the calculation of the inter-pedestrian forces. This implies that the next time step cannot begin before the calculation of the



(a) The floor field like it is used by the router. Since the distance to walls is not considered, the trajectories are partially identical to walls.



(b) The floor field like it is used by the direction strategy. When the propagation of the floor field is slowed down near the walls and obstacles, pedestrians try to avoid them, resulting in a more realistic vector field.

Figure 1.1: A visualization of a floor field for reaching the right side of the room. The color code represents the cost, black lines connect points of equal cost. The black arrows indicate the gradient. The white lines are the trajectories of two pedestrians in the room. The blue U-shape represents an obstacle. Source: [2]

last pedestrian from the previous time step has finished.

All of these tasks are not trivial. While the router could take the linear distance between two doors, this would underestimate the real distance in non-convex rooms. A simple direction strategy could lead the pedestrian directly towards the door. Again, this method is suboptimal in non-convex rooms. The pedestrian might even get stuck when he had to go around several corners or obstacles. A solution to these problems is the floor field, as first described by Burstedde *et al.* in 2001 [1]. Such a floor field (requiring the domain $\Omega \subset \mathbb{R}^n$, e.g. a room, a target $\partial\Omega$, e.g. a door, and a function $f(x)$ with positive values) assigns a value $c(x)$ to every point $x \in \Omega$. This value describes the cost to reach this point from the target (or vice versa). Described by the Eikonal equation, a well known approximation for wave propagation,

$$|\nabla c(x)| = \frac{1}{f(x)}, \qquad \text{subject to} \qquad c|_{\partial\Omega} = 0,$$

the floor field is calculated by the fast marching method [4]. The space domain is discretized. The function $f(x)$ has the physical meaning of the speed of the wave, which corresponds to the maximum walking speed of a pedestrian at the location x in our use case.

Since the router and the direction strategy have different requirements, different floor fields are used. The router needs the distances between any two doors, therefore many floor fields have to be calculated. To compensate for that, a large grid spacing is chosen and $f(x) = 1$ over the whole domain, implying a uniform wave speed. The resulting floor field for a sample geometry is depicted in figure 1.1a. For the direction strategy, the direction a pedestrian should take at a given point needs to be calculated from the gradient of $c(x)$; therefore, a finer grid spacing is needed. Some operational models might have problems with pedestrians being too close to a wall. Under rare circumstances, such a pedestrian might erroneously be treated as being stuck. But forcing the pedestrians away from the wall does not only resolve this computational problem: It can also be observed in reality that pedestrians tend to avoid walls. This can be easily modelled by decreasing the propagation speed of the wave $f(x)$ near walls

and obstacles (see figure 1.1b), as proposed in [2]. This causes the gradient not to point directly to the target or a corner that has to be passed to reach the target, but to also include a component away from the wall, causing pedestrians to leave the wall if they are standing or walking near one and not to approach other walls too close. The resulting trajectories are no longer the shortest way, but the fastest, according to the walking speed $f(x)$. The walking speed close to the walls is not modified in the router where approximate results are sufficient.

The calculation of the floor fields is a computationally expensive task. In order to achieve real-time simulation, parallelization suggests itself. In JuPedSim, this is done by using the directives allowed by the OpenMP standard. They enable us to simply use multiple threads and shared memory.

2 Implementation

2.1 Router Initialization

The router needs to be initialized before it can fulfill its task of leading the pedestrians through the building. For the floor field router, this includes the calculation of all possible floor fields and the determination of the door–door distances (which are saved in a paths matrix) within one room. Afterwards, the Floyd–Warshall algorithm acts on this matrix to find the shortest distance between any two doors.

The calculation of the floor field is an expensive operation. The program logic was in a way that all rooms were calculated in parallel. The problem with this approach is that the floor fields in one room, but belonging to different doors, are calculated one after the other. This is not strictly necessary since these floor fields are independent from each other. The new approach is to collect all pairs of rooms and doors first and then to calculate these in parallel. This leads to smaller chunks of work that can be more uniformly distributed over the available threads. This also tackles the problem of big rooms having many doors, which are two reasons for a long processing time. Since the floor fields for different doors are now split up, only the problem of having big rooms remains. However, splitting up one floor field to calculate it in parallel seems impractical.

2.2 Direction Strategy Floor Fields

Since the direction strategy needs to deliver an accurate direction, a much finer grid is needed than for the router. It is thus highly desired not to calculate more floor fields than needed and – since the calculation takes a considerable amount of time – to structure the program in such a way that other threads do not idle longer than necessary when one thread is calculating a floor field. In this regard, two issues have been identified.

First, there are cases where the same floor field is requested by two pedestrians during the same time step. This case had no special handling and therefore, the calculation took place twice. This does not only waste CPU cycles, but also prevents the thread from calculating other pedestrians, which might need a new floor field, too. The solution to this problem is to keep track of which floor fields are being calculated and not to start the calculation a second time. Instead, the method returns a special value which can be interpreted differently by the operational models. For the models currently implemented in JPScore, this means that the pedestrian keeps its current direction, and possibly slows down a little. Immediately afterwards, the next pedestrian can be calculated by this

thread. In the next time step, the floor field in question will have been calculated by the first pedestrian who needed it and can thus be used by all the pedestrians. It has been verified with different sample geometries that the loss of accuracy in one time step has no visible impact on the overall result (neither on the trajectories nor on the evacuation time).

Second, the calculation of floor fields in a simulation step slows down the whole simulation if this is the last task to finish within one time step. It would be more efficient to calculate the floor fields for the direction strategy during the initialization phase. The needed floor fields have to be identified first; this is currently only implemented if the floor field router is used. This router can provide the direction strategy with a list of all needed pairs of rooms and doors that are presumably needed for the initial pedestrian distribution. The corresponding floor fields can the be calculated in parallel before the first simulation time step.

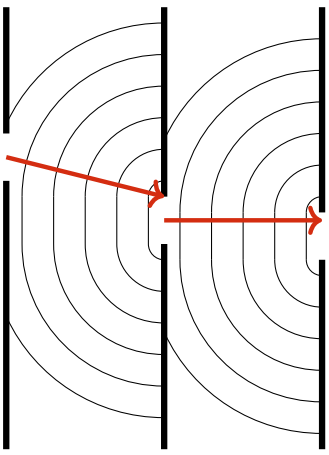
2.3 Critical Constructs

When it comes to parallel programming, it is sometimes necessary to access global variables or class members that are shared among multiple threads. If this happens in an uncontrolled manner, data races may occur, resulting in data corruption or data loss. By the use of a critical construct, OpenMP allows the programmer to restrict the access to such instructions. While a thread is operating within a critical region, no other thread can enter a critical region with the same name.

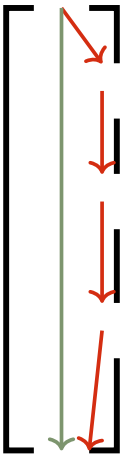
In the code of JPScore, all critical constructs were unnamed. This is of particular interest in the calculation of the floor field. Here, two different class members are accessed several times. While they are independent from each other, this is not properly represented in the code by naming the corresponding constructs. Since threads have to wait at the beginning of a critical region, this increases the wall time of the program and can even lead to deadlocks in some cases. In all known cases, the stalling could be avoided by naming the critical constructs appropriately.

2.4 New Router Floor Field

The router and the direction strategy used the same type of floor field – one that has the whole length of the door as a target. This is desirable for the direction strategy because it allows two pedestrians to walk through this door side by side. However, when used for the router, this might produce unwanted results, as visualized in figure 2.1a. The floor fields for reaching the right and the middle door, which is represented by lines of equal costs, are shown. The distance between two doors is the cost of one floor field, evaluated at the center of the other door. If a pedestrian was to walk this way, he would follow the gradient, as indicated by the red arrows. As is clearly visible, a (hypothetical) pedestrian walking from the left to the right door would not necessarily pass by the center of the middle door, meaning that the length of the path as estimated by the router is shorter than the way the pedestrian would be going to take (notice the small gap between the two arrows at the middle door). The problem that is caused by this becomes obvious if we consider a corridor that a pedestrian wants to walk through from north to south, with doors on the side of the corridor. As depicted in figure 2.1b, the router assumes that the path to the first door on the side of the corridor, then to the next door, and so on until the end of the corridor is shorter than the direct path because



(a) A visualization of the floor fields emerging from the right and the middle door into the room to their respective left side. The red arrows indicate the path from the center of a door to the one to its right, following the gradient of the floor field.



(b) The same sketch for a different room. For clarity, the floor fields are not visualized and the red arrows have been shifted to avoid overlapping with the wall. The route indicated by red arrows is wrongly assumed to be shorter than the direct path (green).

Figure 2.1: Explanation why door hopping happens and its consequences

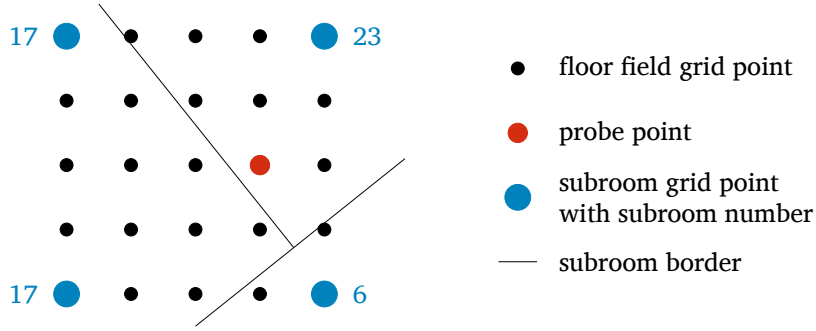


Figure 2.2: A depiction of how the coarse grid for the subroom determination works. The black dots represent grid points of the floor field grid (for clarity, there are only 16 and not 256 floor field grid points per subroom grid point), the larger blue points in the corners are grid points of the newly implemented subroom grid. They store the subroom they are in, indicated by the number. The straight lines are the borders between the subrooms. The red dot is a probe point. Further explanation on how the algorithm operates can be found in the text.

the pedestrian is assumed to skip half the door width at every door. Without further corrections, a pedestrian would either get stuck at a door or walk to the first door, but then walk back into the same room to reach the next door – a behavior that we named "door hopping".

The problem was dealt with in a way such that whenever a pedestrian needed to find the next exit door, the router would check whether the pedestrian leaves the room through this door, and would return this door only if this condition holds. However, when it comes to getting the presumable exit route (to initialize the floor fields of the direction strategy), the same problem arises again: Without further correction, the floor fields belonging to the side doors are calculated as well as the floor field for the final door, although the first ones are not needed.

To overcome this problem, a new kind of floor fields has been implemented for the router: a floor field that leads only to the center of the door. Now, the distance between two doors is always calculated as the distance of the two center points. These distances satisfy the triangle inequality, which avoids door hopping completely and is therefore well suited for the router. It is not advisable to use this kind of floor field for the direction strategy because it effectively shrinks the door to a point, allowing only one pedestrian at once to pass through it.

2.5 Determination of the Subroom

For the calculation of the floor field, it is necessary to know whether a given point of the floor field grid is inside or outside of the room. Since a room is not aware of its boundary, this is achieved by iterating over all the subrooms (disjoint subunits whose union equals the whole room) and checking whether the probe point lies within this subroom. Doing this for every grid point (the currently implemented spacing is 6.25 cm) is computationally expensive.

The implementation has been changed to do this expensive calculation only for the grid points (blue in figure 2.2) of a coarser grid (1 m in the current version), named

"subroom grid" below. There are thus 256 grid points of the floor field grid belonging to a square formed by four points of the subroom grid. When the subroom of a floor field grid point is needed, the subrooms in which the four corners lie are checked first. Only if the check in all (up to four) subrooms is unsuccessful, the complete list of subrooms (excluding the ones already investigated) is checked. If in figure 2.2, the subroom of the red probe point shall be determined, the subroom of the lower left corner (17 in this case) is checked. Since the red point is not in subroom 17, the next subroom grid point is checked, e.g. the upper left one. Since subroom 17 has already been checked, it is not checked again. Then, the subroom of the upper right corner, subroom 23, is checked. Since this check is successful, the algorithm terminates. While with the old approach, all subrooms from 1 to 23 had been checked, only two subrooms (17 and 23) were checked in the subroom grid approach in this case, which is a significant reduction. Since we still check the subroom for every floor field grid point, it is guaranteed that for every such point, the correct subroom (and therefore the information whether a point is inside or outside the room) is still obtained in every case, so the behavior of the program is not altered.

2.6 Direction Strategy Initialization Scheduling

A lot of the execution time of the program is spent in the initialization of the direction strategy for the creation of the grid. In the investigated sample geometry, this initialization took around 24 s, which is more than half of the complete execution time of the program (around 40 s, all measurements with 8 threads). It is therefore desirable to speed up this part.

The current scheme is that all rooms initialize their grid in parallel, using the *parallel for* directive from OpenMP. There are three different scheduling kinds that can be used together with this directive: *static*, *dynamic*, and *guided*. With the *static* clause, all loop iterations are distributed as evenly as possible over the available threads before the execution of the loop starts. With the two clauses *dynamic* and *guided*, not all the work is distributed at the beginning. Rather, once a thread finishes its current chunk of work, it requests a new one. A *guided* scheduling starts with distributing larger chunks that decrease in size when most loop iterations have been distributed. The idea behind this is to have fewer portions to distribute and therefore less overhead. For all three scheduling kinds, a chunk distributed is never smaller than the chunk size specified (except the last one). The *dynamic* and *guided* scheduling are usually beneficial when the tasks that have to be done are very different in size – as is the case here with rooms of dissimilar areas. For this reason, a significant difference between different scheduling kinds can be expected.

For the aforementioned geometry, the three scheduling kinds have been examined with chunk sizes varying from 1 to 4. In each test, 8 threads have been used. For the *static* kind, an additional test has been run where the chunk size parameter has been left out, causing the chunk size to be determined in such a way that all loop iterations are distributed at once. This is also the default when no scheduling clause is specified at all. For the other two scheduling kinds, the default value is 1. The average of five measurements for each possible combination is listed in table 2.1. As can be seen from these data, the times for the investigated set of parameters all lie between 24.0 s when using *static* with a chunk size of 3 or 4 and 24.4 s when using *dynamic* or *guided* with

Table 2.1: Measured initialization times in s

chunk size	static	dynamic	guided
1	24.1	24.3	24.4
2	24.2	24.4	24.4
3	24.0	24.4	24.4
4	24.0	24.4	24.4
not specified	24.1	–	–

almost every chunk size investigated. It has been observed that the recorded times vary by up to 0.3 s for the same measurement, so it seems reasonable to assume a margin of error of ± 0.2 s. From the data, we can see that the *static* scheduling is faster by 1.6 %. This might be caused by the additional overhead for the *dynamic* and *guided* scheduling. For all scheduling kinds, the difference between varying chunk sizes lies within the assumed margin of error.

3 Scaling

The scaling behavior of the program was analyzed when between 1 and 8 threads are used. The time for the initialization of the direction strategy and the complete execution time of the program were measured for 10 times each, using both the *static* and *dynamic* scheduling kind with their default chunk size. From the values of the *static* scheduling, the duration of the non-initialization part of the program has been calculated. The average values are visualized in figure 3.1.

The data clearly show that the scaling of the program can still be improved. The time needed for the initialization of the direction strategy with *static* scheduling does not scale: It is either 32 s to 33 s if 1, 2 or 4 threads are used, or 24 s when another number of threads (of those tested) are used. Further investigation of this issue showed that this alternating behavior is specific for the used geometry. The reason is an unfortunate distribution of the rooms to the available threads. With 1, 2 or 4 threads, the two biggest rooms are assigned to the same thread (this is due to the order of the rooms and the *static* scheduling that was used), increasing the run time. Since for 8 cores, the rooms are distributed in a favorable way, different scheduling types as tried out in section 2.6 had little influence. Redoing the scaling test with *dynamic* scheduling with the default chunk size of 1 demonstrates the performance gain when 2 or 4 threads are used. For other number of threads, a negligible overhead is introduced. However, the initialization only scales up to 3 threads; from then on, the time is constant between 24.4 s and 24.5 s. The rest of the program is independent of the scheduling kind. For this part, the time reduction when using multiple threads can easily be seen. The serial fraction of the non-initialization can be calculated by linear regression on Amdahl's law; in the investigated case, it is 54 %. This includes, however, the parsing of the input files (configuration and geometry), a task that cannot be parallelized.

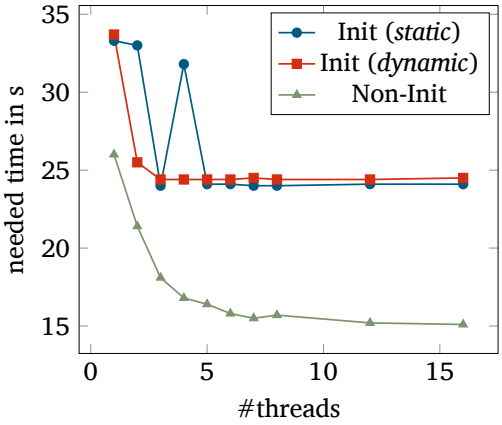


Figure 3.1: Scaling for the optimized program using a sample geometry. Shown are the times the initialization (abbreviated "Init") of the direction strategy needs with two different scheduling kinds as well as the rest of the program ("Non-Init").

4 Conclusion & Outlook

In this report, several improvements to the JPScore modul of JuPedSim have been presented. The main task was to speed up the calculation of the parts of the code involving floor fields, namely those routers and direction strategies who are based on them. For some of the changes, a performance improvement could be measured. Not all modification already develop their full potential though. It is to be expected that the further application of JuPedSim (e.g. on bigger buildings) will make greater demands on the program, making parallelization even more important.

In section 2.4, a new kind of floor field that has been implemented was presented. The door hopping problem has been a known issue that was dealt with in a working, yet unsatisfying way. With the changes made (in this case, shifting the calculation of the floor fields for the direction strategy to the initialization phase), this issue came up again and the inconvenience emanating from it grew. The floor field starting only at the centre of the door does not only overcome the aforementioned issue, but also adds to the variety of models and options JuPedSim offers.

For the initialization of the direction strategy, the most time consuming task of the whole program, different scheduling clauses (*static*, *dynamic*, and *guided*) for the OpenMP pragma have been tried out. When using 8 threads, no performance gain was observed, but the additional overhead introduced by *dynamic* and *guided* scheduling was measured. We soon realized that the scheduling only had an influence under certain circumstances. For the investigated geometry, using 8 threads does not yield such a case, but using 4 threads does. In this case, the dynamic scheduling does indeed reduce the initialization time, as we expected. However, the time does not change anymore when more than three threads are used. A possible reason is the large amount of memory that is allocated, but this needs further investigation. If this is indeed the cause, the solution is to reorganize the memory allocation and its use.

JuPedSim is still under heavy development: New methods, models and features and

added, old ones are improved or removed if they are rarely used in practice because they are inefficient or yield results that model the reality insufficiently. Since new ideas come up all the time, structures that have been in the program code for a long time are not safe from being reworked. One such example is how the subroom is determined. The implementation of a coarser grid turned out to be a success. It is obvious that a similar concept can be applied to other grid data used in the program, e.g. the distance to walls.

Acknowledgement

First of all, I would like to thank my supervisors Arne Graf and Mohcine Chraibi for giving me the opportunity to work on JuPedSim and to contribute to it. They were a great guidance and very helpful for all the obstacles I encountered. I am also very grateful to the JSC and the organizers of the Guest Student Programme, especially to Ivo Kabadshow. Last but not least, I wish to thank all my fellow guest students for the great time we spent and the activities we made together. Without them, the program would have been much less fun.

References

[1] C. Burstedde, K. Klauck, A. Schadschneider, and J. Zittartz. **Simulation of pedestrian dynamics using a two-dimensional cellular automaton**. *Physica A: Statistical Mechanics and its Applications*, 295:507 – 525, 2001. doi:10.1016/S0378-4371(01)00141-8.

[2] A. Graf. **Automated Routing in Pedestrian Dynamics**. Master's thesis, FH Aachen, 2015. http://juser.fz-juelich.de/record/276318.

[3] A. U. Kemloh Wagoum, M. Chraibi, and J. Zhang. **JuPedSim: an open framework for simulating and analyzing the dynamics of pedestrians**. In *3rd Conference of Transportation Research Group of India*, 2015 (To appear).

[4] J. A. Sethian and J. A. Sethian. *Level set methods and fast marching methods: evolving interfaces in computational geometry, fluid mechanics, computer vision, and materials science*. Number 3 in Cambridge monographs on applied and computational mathematics. Cambridge University Press, 2nd ed edition.

A Task-Based Approach to Parallelize the FMM

Dynamic Load-Balancing through Work-Stealing

Laura Morgenstern
B.Sc. Applied Computer Science
Chemnitz University of Technology
Germany
lmor@hrz.tu-chemnitz.de

Abstract *The Fast Multipole Method (FMM) is a fast summation technique for computing the long- and short-range interactions in particle systems. Based on a modern, sequential C++ implementation of the method we present and analyze a task-based intra-node parallelization approach by means of `std::thread`. In the course of that, we utilize the concept of work-stealing as dynamic load-balancing technique.*

1 Introduction

The simulation of interactions between particles is of peculiar interest in diverse research areas such as astrophysics, plasma physics and molecular dynamics. These areas have in common the necessity to describe the motions of numerous objects interacting with each other. In order to accomodate this necessity by simulating dynamic particle systems, long-range Coulomb interactions have to be considered. Due to the absence of an analytical solution for this so called N-body problem for $N \geq 3$, where N is the number of particles, we have to compute the acting forces numerically, in order to obtain the resulting motions. To compute the force acting on one particle, a classical Coulomb solver would ascertain the interaction between this particle and each of the remaining particles, which leads to $\mathcal{O}(N)$ computations. Due to the fact that we need to determine a force vector $\boldsymbol{F}$ for each particle in the system, this method leads to a computational complexity of $\mathcal{O}(N^2)$. Considering that realistic systems may contain millions of particles and that each timestep of a simulation costs $\mathcal{O}(N^2)$ computations, such a direct approach is not feasible.

Therefore the development of more efficient methods, e.g. fast summation techniques, was of great interest. One of these fast summation techniques is the fast multipole method (FMM), which was developed by Rokhlin and Greengard in 1987 [4]. The FMM reduces the complexity of computing the long-range interactions from $\mathcal{O}(N^2)$ to $\mathcal{O}(N)$ by spatial grouping of particles, based on the idea, that particles further away from an oberserved particle affect that particle less than particles in the proximity.

Since the theoretical concepts of the FMM need to be implemented for todays HPC systems, we outline the initial requirements briefly. Foundation of this work is a highly templated `C++11` implementation of the mentioned fast multipole method. Since the implementation already covers inter-node parallelization by means of MPI, the goal of this work is to describe an intra-node parallel implementation with `std::thread`. This is necessary to advance towards the goal of simulating one timestep for an arbitrary sized particle ensemble in only one millisecond. Due to that goal, the parallel implementation not only needs to cope with as few particles per node as possible, but also needs to fullfill the requirements of strong scaling – even on many-core architectures like Intel's Knights Landing [5].

2 Essentials

In this section we first depict the workflow of the FMM as it is specified in reference [7]. In the course of this, we leverage the methods parallelization potential. Subsequently, we introduce the concepts of tasking, in order to combine these two in section 3.

2.1 Workflow of the Fast Multipole Method

As input the FMM receives the coordinates $\boldsymbol{x}$ and the charge q of each particle in the system. The workflow of the method, starting with this input, is depicted in figure 2.1. Output of the FMM are the forces $\boldsymbol{F}$, the potential ϕ and the Coulomb energy E of the system. Based on the particles force vectors, velocities and corresponding position updates for the particles can be computed. Subsequently we will describe how to transfer input data into output data step by step. In a preprocessing step, we first have to determine three parameters. Firstly, the multipole order p, which determines the precision of the computed results. Secondly, the well-seperateness criterion ws, which differentiates a boxes environment into near- and farfield. In the implementation ws is configureable, but in this work $ws = 1$ applies constantly, because this reduces the number of direct interactions to a minimum and yields the best performance. Thirdly, we determine the maximal tree depth d_{max} of the derived FMM tree with regard to the multipole order p and the number of paticles N.

The principle mentioned in section 1, that remote particles affect a particle less than particles in the proximity, and its consequence – the possibility to group remote particles together in form of pseudo-particles – is illustrated in figure 2.2. The illustrations depict the inter-cluster interactions between two groups of particles and exhibit, that spatial grouping leads to a reduced number of interactions.

Based on this concept, we start out with a hierarchical space subdivision of the cubic simulation box. We do so by bisecting the cube in each dimension, which generates eight equally sized child boxes. For a three-dimensional particle system this results in a data-structure commonly referred to as octree. The subdivision of the child boxes is recursively continued until the determined tree depth d_{max} is reached. Figure 2.3 illustrates this principle for a two-dimensional system. The levels of the tree can be named by depth d, whereas $d = 0$ indicates the root of the tree, respectively level index l, whereas $l = d + 1$.

Having finished the initial setup, we will now go through the five passes of the FMM, depicted in 2.1, with respect to parallelization concepts. In passes 1 to 4 the far field contributions are computed, while the near field contributions are independently computed in pass 5.

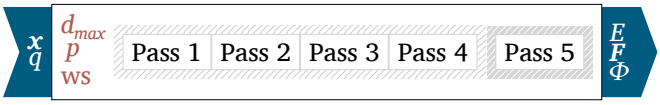
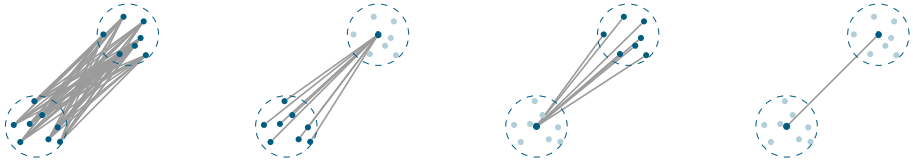


Figure 2.1: Workflow of the FMM.

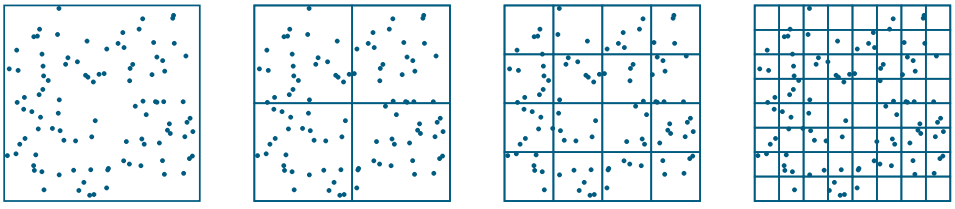


(a) (b) (c) (d)

Figure 2.2: Spatial grouping of particles in order to reduce the number of interactions, see [3] for details (picture source: [3]). 2.2a Direct interactions between particles, except inter-cluster interactions. 2.2b Grouping particles of upper cluster to pseudo-particle; interaction between pseudo-particle and particles in the lower cluster. 2.2c Grouping particles of lower cluster to pseudo-particle; interaction between pseudo-particle and particles in the upper cluster. 2.2d Interaction between pseudo-particles.

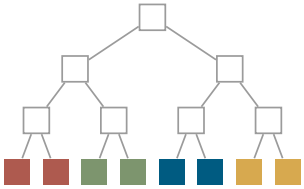
2.1.1 Pass 1: Particle to Multipole (P2M) & Multipole to Multipole (M2M)

Pass 1 consists of two steps. The first step is the expansion of the particles on the lowest level into multipole moments, referred to as P2M. Assuming that we have a number of threads t, e. g. $t = 4$, this step could for instance be parallelized as illustrated in figure 2.4a. The figure shows a binary tree for illustration. Each color stands for one thread, which means that each thread could perform P2M for two boxes independently from the other threads. Generally speaking each thread could perform P2M on $8^{d_{max}}/t$ boxes. Due to the fact, that the boxes do not necessarily cover the same number of particles, this approach would already lead to load-imbalances. Another parallelization approach would be to distribute particles equally to the threads. The latter would admittedly lead to a perfect load-balancing for this step, but would conversely introduce synchronization and further parallelization overhead. Synchronization overhead due to the fact, that multiple threads construct the multipole of the same box; further parallelization overhead, due to a larger number of work units to be scheduled. In the second step of pass 1 we translate the constructed multipole moments ω up the tree. As depicted in figure 2.4b, this step could be performed on subtrees partly in parallel. Starting on the lowest level $l = 4$ each of the four threads could shift two multipole moments from the lowest level to the center of the corresponding parent box. Subsequently each thread could translate the so created parent boxes multipole moment

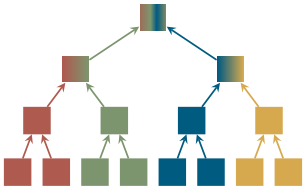


(a) $d = 0$ (b) $d = 1$ (c) $d = 2$ (d) $d = 3$

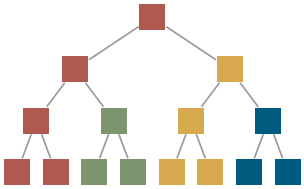
Figure 2.3: Hierarchical space subdivision of a two-dimensional system, see [3] for details (picture source: [3]).



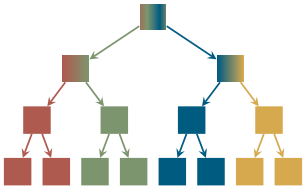
(a) Parallelization of P2M, L2P and P2P.



(b) Parallelization of M2M.



(c) Parallelization of M2L.



(d) Parallelization of L2L.

Figure 2.4: Pass-dependent parallelization approaches.

one more level up the tree. As the color gradients in figure 2.4b indicate, we need the threads to synchronize at these points. Due to the $\mathcal{O}(N)$ requirement of the FMM, we cannot shift the multipole moment one level up, before the last child box multipole moment was shifted to the center of the parent box. In consequence of the mentioned synchronization points, respectively the tree getting smaller with decreasing depth d, an increasing number of threads becomes idle while going up in the tree towards the root. With respect to strong scaling this causes trouble, because – as Amdahl's law implies, the theoretical speedup is limited by the sequential parts of a program. Meaning, as long as we have to do at least something sequentially, e.g. the M2M operation for the root node, we will not reach ideal scaling.

2.1.2 Pass 2: Multipole to Local (M2L)

In pass 2 we transform the multipole moments ω on each level into local moments μ. An approach to execute these transformations in parallel is shown in figure 2.4c. M2L needs to be executed for each box and the operations do not depend on each other. Hence we can distribute the work as follows: Starting from the lowest level, an equal amount of boxes is assigned to each thread. Regarding the example in figure 2.4c this would correspond to two boxes per thread. Each time we go up one level, we have to decide to which thread the M2L operation of the parent box is assigned. In the example this operation is by convention assigned to the thread, which performed M2L on the left child. As the example exhibits, this work distribution leads to load-imbalances due to the tree structure. However, the described distribution also leads to load-imbalances for two more reasons. Firstly, the tree may contain empty boxes for which no M2L operation needs to be done. As a consequence, the threads to which these boxes are assigned, have less work. Secondly, in a not fully periodic system, the boxes located near the boundaries have less neighbors. Therefore, the threads to which these boxes are assigned, have to compute less local moments μ.

2.1.3 Pass 3: Local to Local (L2L)

In this step we start at the root and shift local moments μ down the tree. Figure 2.4d depicts a possible parallelization approach. Starting at the root box each thread could shift the local moment of the root box to an equal amount of child boxes. In the shown example we spawn two threads to do so. Once the local moment has reached the child box, we could again spawn threads and assign an equal amount of shifting-operations to them. Generally speaking, we could make use of 8^d threads to shift the local moments from depth $d-1$ to level d, assuming that each thread executes one L2L operation. Applying this principle until the local moments reach the lowest level, it becomes apparent, that parallelism increases from the top to the bottom of the tree.

2.1.4 Pass 4: Local to Particle (L2P)

This pass covers the computation of the far field contributions by translating the local moment of each box on the lowest level to each particle in the according box. Figure 2.4a points out, that L2P can be parallelized in a similar manner as P2M. Each of the t threads performs L2P for all particles in $8^{d_{max}}/t$ boxes on the lowest level. Regarding the example, each of the four threads performs L2P for two boxes.

2.1.5 Pass 5: Particle to Particle (P2P)

So far we have described the passes required to compute the far field contributions. In order to complete the computation of the interactions, in pass 5 the near field contributions are computed. Thus the pairwise interactions between the particles in one box on the lowest level and the particles in the adjacent boxes, with respect to the well-seperateness *ws*, are taken into account. Even though we refer to this pass as pass 5, it has not necessarily to be executed after the abovementioned passes. Thanks to being independent from the other passes, P2P can even be computed in parallel to all other passes. Furthermore, P2P itself can be parallelized analogous to L2P respectively P2M as shown in figure 2.4a.

2.2 Tasking

In addition to the threading-concept, applied in section 2.1, we introduce the concept of tasking in this section.

We refer to tasking as the assignment of tasks to threads. In this context a task is a unit of work of specified size, whose execution contributes to a major computation.

To give an example for tasking, we consider the subdivision of a classical Coulomb solver into tasks. One way to do so, would be to consider the computation of a single interaction between two particles out of all pairwise computations as a task, which would lead to small tasks and hence a fine granularity. However, a task could also be the computation of the interaction between a certain particle and all other particles, which would lead to a much coarser granularity. Thanks to the tasks being independent from each other no matter which subdivision we choose, the tasks can be equally assigned to the threads and be computed in parallel.

3 Taskifying the Fast Multipole Method

After introducing the workflow of the FMM and the basics of tasking, we will now combine these two. To do so, we could, sorted by ascending granularity, subdivide the algorithm into tasks based on levels, boxes or particles. In the subsequent sections we

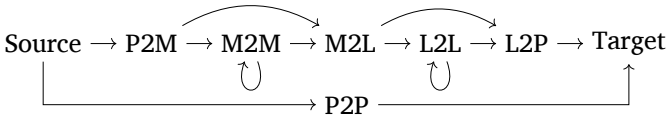


Figure 3.1: Dependencies between tasks. Dependencies that need to be resolved to transform a source particle into a target property.

stick to boxes in terms of the granularity level, because the tasks are large enough, so the distribution overhead does not prevail. However, we will introduce load-imbalances by sticking to boxes, e.g. due to the boxes covering different amounts of particles. Due to these load-imbalances, we will also present a dynamic load-balancing approach in this section.

3.1 Tasks

Based on the workflow of the FMM, we differentiate six types of tasks – P2M, M2M, M2L, L2L, L2P and P2P. Each P2M task covers expanding all of the particles into a multipole in a single box on the lowest level. From the perspective of a single parent box an M2M task translates the multipole moments of all its eight child boxes to its center. In a three-dimensional system each M2L task comprises up to 189 M2L operations. An L2L task covers the translation of the local moment μ of a specific parent box to local moments in its eight child boxes and computing the farfield forces. Each L2P task involves shifting the local moment of a box to all the particles in the according box. With a single P2P task the near field contributions for all of the particles in one box with respect to the boxes well-seperated neighbors are computed.

3.2 Task Dependencies

Having described the tasks we can now consider the task-dependencies arising from the abovementioned subdivision as shown in figure 3.1. In the current implementation these dependencies are resolved through the fact, that tasks generate their succeeding tasks by means of dependency counters. Once being generated, each of the tasks needs to be assigned to a thread. This is implemented by storing the generated, but not yet performed tasks in a data structure referred to as a multi-queue.

3.3 Multi-Queue

To store tasks in the order of their creation and execute them in the same order, a FIFO-queue would be sufficient. However, since we have different task types, we need a data structure resembling this circumstance. Hence, we designed and implemented a class `MultiQueue`, which initially provides a single queue for each task type. Each of the six task queues is implemented by `std::deque`, the double-ended queue structure from the `C++` standard library. This enables threads to take tasks from the front and from the back of the queue. Since `std::deque` is not thread-safe, each of the single queues is protected by a `std::mutex` in case a thread adds or removes a task from the according queue.

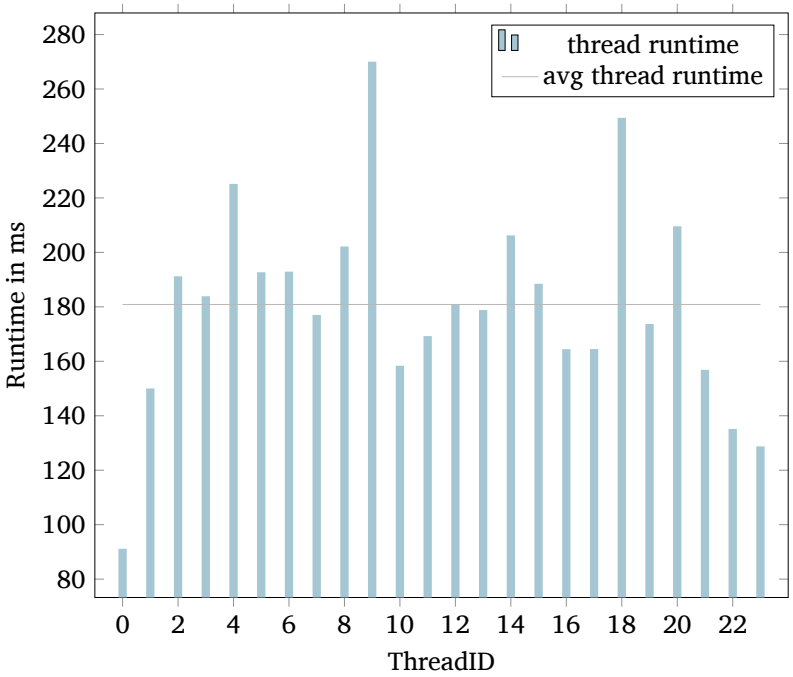


Figure 3.2: Workload per thread without work-stealing.

3.4 Work-Stealing vs. Work-Sharing

There exist two contrasting concepts to dynamically schedule tasks in a multi-threaded application – work-sharing and work-stealing. Work-sharing is based on a global data structure containing all of the tasks, which can be executed right away. Here the scheduler tries to distribute the arising work as equally as possible among the threads. In a work-stealing approach each thread has got its own data structure, which stores the tasks it has to perform. Since there is no scheduler assigning tasks to threads in work-stealing, threads autonomously try to steal tasks from other threads when running out of work. In contrast to work-sharing, work-stealing scales, because there is no central scheduler, which all of the threads depend on. Another advantage of work-stealing in comparison to work-sharing is that tasks only migrate from one thread to another if necessary. Thanks to these advantages, the subsequently described idea is based on the first feasible work-stealing scheduler for multi-threaded computations with dependencies presented by Blumofe, Leiserson and Charles in reference [1].

Plot 3.2 emphasizes why we in fact need work-stealing. The plot depicts that the runtimes of the threads vary widely. In order to equalize the runtimes, threads with less work, in the plot e.g. thread zero, should steal work from threads with more work, e.g. thread 9. This could be done as shown in listing 3.1.

Listing 3.1: Work-Stealing

```
1  void RunOne()
2    {
```

```
3         unsigned int num_threads = queues.size();
4         std::pair<std::function<void(void *)>, void *> work_unit(
5                 [](void *) {}, nullptr);
6         if (!queues[main_que_index]->empty())
7         {
8             work_unit = queues[main_que_index]->take_front();
9         }
10        else if (stealing)
11        {
12            unsigned int i = (main_que_index + 1) % num_threads;
13            while (!queues[i]->stealable() && i != main_que_index)
14            {
15                i = (i + 1) % num_threads;
16            }
17            if (queues[i]->stealable())
18            {
19                work_unit = queues[i]->try_steal();
20            }
21        }
22        work_unit.first(work_unit.second);
23    }
```

First, we retrieve the number of threads, respectively multi-queues, in line 3. Second, a work unit (which conforms a task) is initialized with an empty lambda-function in line 4. This is done so that in any case a work unit can be executed at the end of `RunOne()` in line 22. Next it is checked, whether the threads multi-queue is empty. If the threads own queue is empty, we step into the `else`-branch in line 10 and try to steal a task, respectively a work unit, from another threads multi-queue. Starting with `thread_id+1`, we iterate ring-like over all multi-queues until we either reach `thread_id-1` or find a multi-queue to steal from. To check whether stealing from a queue is possible, the function `stealable()` is used; `stealable()` returns `true`, if the according queue contains more than one work unit. `stealable()` is not thread-safe and therefore only returns a hint, that there could be a task to steal. Hence, we can only try to steal from the ascertained queue by means of `try_steal()`. If `try_steal()` indeed returned a task from another threads queue, we then execute the stolen task in line 22. Otherwise an empty lambda-function – either returned from `try_steal()` or from the work unit's initialization – is executed. Plot 3.3 contrasts the workload of each thread with and without the presented work-stealing approach.

The plot depicts that work-stealing leads to an equal load-distribution, hence a decreased parallel runtime. However, in comparision to the average runtime without stealing, we introduced an overhead of 4.4% through work-stealing.

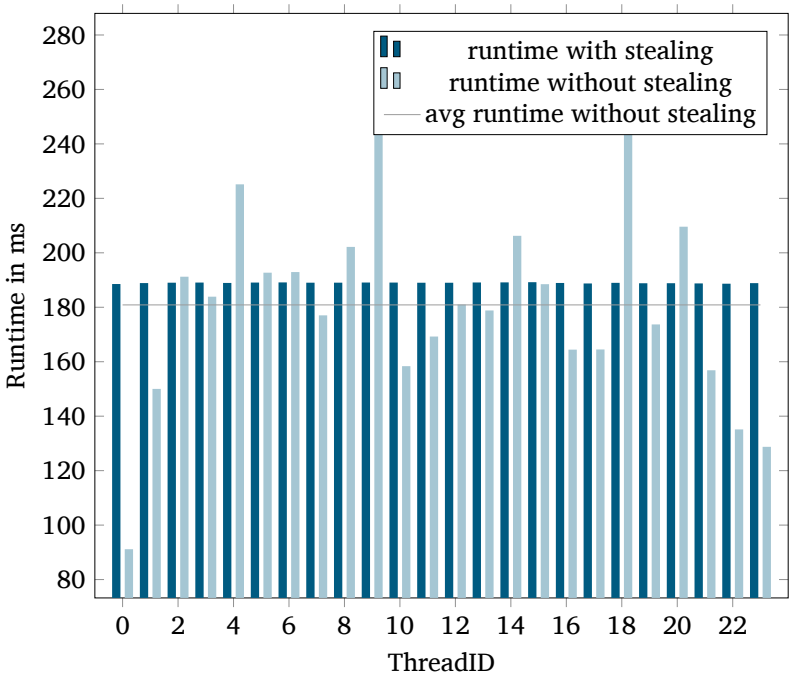


Figure 3.3: Workload per thread with vs. without work-stealing.

4 Performance Analysis

In this section we analyze to which extent a better load-balancing through work-stealing improves not only the parallel runtime, but also the efficiency respectively the scaling of the implementation. Let's introduce our measuring technique first.

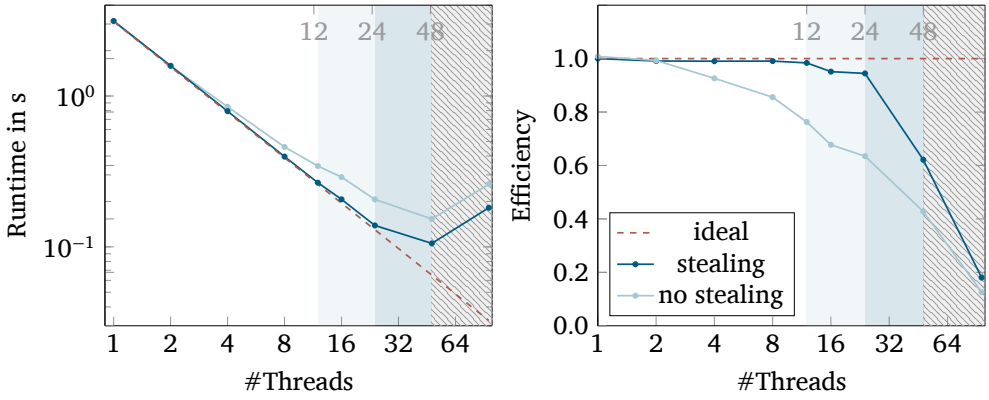
4.1 Measuring Technique

The time measurements for all of the presented plots are done on a single compute node of JURECA [6]. Each compute node is equipped with two Intel Xeon E5-2680 CPUs, conforming 24 physical cores and 48 logical cores. The measurements are performed by `std::chrono::high_resolution_clock`, which on JURECA leads to a resolution of one nanosecond. In order to get reasonable timing results, we execute the application 500 times with the small input data set (1k particles) and 50 times with the large input data set (100k particles). To avoid the influence of clock frequency variations during the starting phase of the measurements, only the runtimes of the last 30% are taken into account to average the runtime. Turbo boost is disabled during the measurements, because it leads to varying clock frequencies i.a. depending on the number of used cores. Due to this, scaling plots can be distorted. To which extent turbo boost influences runtime is described by Charles, Jassi et al. in reference [2]. Since JURECA's compute nodes are non-uniform memory access (NUMA) systems, with 12 cores per NUMA node, a pinning policy needs to be implemented. For measurements with up to 12 threads, we use a compact pinning policy, utilizing only one NUMA node. We do so in order to avoid the influence of data exchange between the nodes. For larger thread counts, NUMA effects can not be avoided. Hence, threads are distributed among the two nodes automatically by the operating system without manual pinning.

4.2 Large Input Set

In this section we analyze the performance of the implementation on a homogenous system with 100k particles. Figure 4.1a shows the according scaling plot. The light blue line shows the runtime for the implementation without work-stealing, while the dark blue line depicts the runtime with work-stealing. The dashed red line represents ideal scaling. The light blue background depicts the number of threads from which on threads are pinned to the cores of both NUMA nodes. The darker blue area indents the usage of simultaneous multi-threading (SMT), while the gray area conforms the area beyond SMT. As the plot illustrates, both the versions scale up to a number of 48 threads, which notably means that the implementation even scales when using SMT. However, the curve of the version with work-stealing runs much closer to the optimal runtime than the curve of the version without dynamic load-balancing. This shows that the administration overhead introduced through the dynamic load-balancing is smaller than the runtime improvement precipitated through it. Furthermore the plot depicts that there is no more runtime improvement when increasing the number of threads further. The reason for this is, that the compute nodes only support 2× SMT, which means that we experience costly context switches when using more than 48 threads. The subsequent plot 4.1b shows the related efficiency plot.

The efficiency plot emphasizes once more that the load-balanced version scales better than the non-balanced version. However, an efficiency decline becomes apparent as soon as more than 12 threads are used. The reason for this is, that we have to use



(a) Scaling plot

(b) Efficiency plot

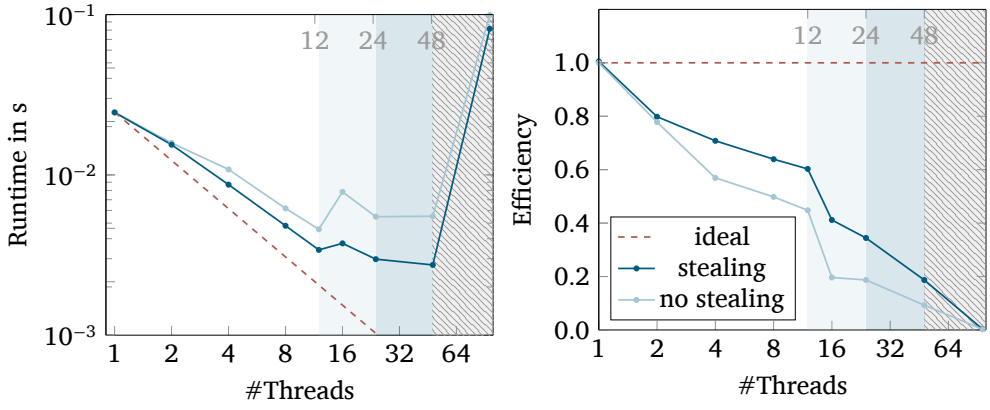
Figure 4.1: 100k particles, multipole order $p = 10$ and tree depth $d_{max} = 4$.

both NUMA nodes when exceeding a quantity of 12 threads without considering this in terms of memory management. Meaning, that the threads running on NUMA node 1, in order to get the data to compute on, permanently need to request memory access from NUMA node 1, which leads to higher task execution times and in turn to a higher parallel execution time.

4.3 Small Input Set

In analogy to the previous section we subsequently analyze the runtime behavior of a smaller particle system covering only 1k particles by means of plot 4.2a. The line and area colors used in the plot have the same meaning as previously described.

Even though load-balancing through work-stealing leads to a runtime improvement independently from the number of used threads here too, the curves are by far not that close to the ideal curve as for the large particle system. A possible reason for this could be that hiding of memory access latencies is less effective due to less computational work. This would also explain the fact that the efficiency decline due to the usage of two NUMA nodes becomes more apparent in the scaling and efficiency plots for the small particle system. The reason for the lower computational effort for one thing is the lower amount of particles and for another thing the lower multipole order. The efficiency plot in figure 4.2b depicts the massive efficiency decline even more clear.



(a) Scaling plot

(b) Efficiency plot

Figure 4.2: 1k particles, multipole order $p = 3$ and tree depth $d_{max} = 3$.

5 Conclusion & Future Work

We described a task-based approach for an intra-node parallelization of the FMM by means of `std::thread`. In the course of that, we presented an FMM-aware multi-queue and a work-stealing implementation to equally distribute tasks among threads dynamically. Considering the small input set with 24 threads, our intra-node parallelization with work-stealing led to an overall parallel runtime of 2.7 ms. In comparison with the sequential runtime of 24.6 ms this leads to a speedup of 9.1 and brings us closer to the 1 ms goal.

However, we have not yet achieved it. Due to the observed efficiency decline (cf. 4.2 and 4.3), the implementation of a NUMA-aware memory management is next up on our agenda. In the course of this the work-stealing approach also needs to be adapted to the NUMA topology, e.g. by preferably stealing tasks from threads running on the same NUMA domain. So far the tasks are taken from the `MultiQueue` and executed in the order according to the sequential FMM's workflow without considering the critical paths through the tree. Due to this another step is the implementation of a priority multi-queue with potentially dynamically adaptable priorities to execute tasks on the critical paths preferential. Especially through the latter we gather knowledge about task dependencies and task granularity in order to advance towards a static load-balancing through FMM-aware task distribution.

6 Acknowledgment

To begin with, I am grateful to my supervisor David Haensel for patiently answering my questions regarding e.g. the `C++` implementation in general and template meta programming in particular. Furthermore I would like to give thanks to Ivo Kabadshow not only for repeatedly explaining the FMM's workflow to me, but also for the night-time programming sessions. In addition I would like to thank Andreas Beckmann for his support including performance measurements and shell scripting. Moreover I gratefully acknowledge the support of Michael Hofmann not merely regarding the application

phase for the Guest Student Programme, but rather regarding the discussion of and his remarks on this report. Thanks go also to Christine Jakobs for calling my attention to the programme and to Thomas Jakobs for his support especially in the application stage. Last but not least, I would like to give thanks to the other guest students for working, travelling and living together.

References

[1] R. D. Blumofe and C. E. Leiserson. **Scheduling multithreaded computations by work stealing**. *Journal of the ACM (JACM)*, 46(5):720–748, 1999.

[2] J. Charles, P. Jassi, N. S. Ananth, A. Sadat, and A. Fedorova. **Evaluation of the intel® core™ i7 turbo boost feature**. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 188–197. IEEE, 2009.

[3] A. García. **FMM rotation operators on speed – Accelerating a Coulomb solver with GPUs**. 2015.

[4] L. Greengard and V. Rokhlin. **A fast algorithm for particle simulations**. *Journal of computational physics*, 73(2):325–348, 1987.

[5] Intel. **Intel® xeon phi™ processor 7210 (16gb, 1.30 ghz, 64 core)**. `http://ark.intel.com/products/94033/Intel-Xeon-Phi-Processor-7210-16GB-1_30-GHz-64-core`, 2016 (accessed October 15, 2016).

[6] F. Jülich. **JURECA**. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/JURECA_node.html`, 2016 (accessed October 15, 2016).

[7] I. Kabadshow. *Periodic Boundary Conditions and the Error-Controlled Fast Multipole Method*. PhD thesis, 2012.

Distributed CPU/GPU parallelization of the ChASE library

Parallelization of the Chebyshev filter

Josip Žubrinić
Faculty of Science,
Department of Mathematics
University of Zagreb
Croatia
josip.zubrin@gmail.com

Abstract *We investigate the possibilities of parallelization of the computationally most intensive part of the ChASE eigensolver. This part, called Chebyshev filter greatly increases the convergence rate of the algorithm. Parallelization is done on multiple levels, involving both multi-GPU and multi-CPU implementations. We propose an alternating-cycle implementation of Chebyshev filter in which we reduce the required communication to the minimum.*

1 Introduction/Motivation

The main topic of this project is the hybrid parallelization of the computationally most expensive part of the iterative eigensolver ChASE. The Eigenproblems are the class of problems which frequently appear in the applications. One of the most common source of algebraic eigenproblems is the discretization of certain partial differential equations.

There are many methods to solve algebraic eigenproblems. We focus on a particular method based on the accelerated subspace iterations. This method incorporates the Chebyshev filter, a particularly useful strategy of acceleration. This strategy involves iterative multiplication of dense tall and skinny matrices which makes it the most expensive part of the solver. The aim of this paper is to optimize a hybrid parallelization of such kernel on distributed CPU/GPU platforms.

However, before we dive into examining strategies, we have to set the theoretical background.

2 Eigenproblems and the ChASE library

We begin with the definition of an eigenvalue of a matrix. A complex scalar λ is an *eigenvalue* of the square complex matrix H if there exists a nonzero vector $x \in \mathbb{C}^d$ such that

$$Hx = \lambda x. \tag{2.1}$$

The set of all the eigenvalues of H is reffered to as the spectrum of H. The vector x is called an *eigenvector* of H. Usually we call the pair (λ, x) *eigenpair*. Every square full rank complex matrix of the size $d \times d$ has exactly d eigenvalues. In the case that all the eigenvalues are distinct, the matrix has also d uniquely distinct eigenvectors.

Eigenvectors make up the subspaces of $\mathbb{C}^d$ called the *eigenspaces*. As we are interested in the Hermitian eigenvalue problems, we will focus our discussion in that direction. The complex square matrix H is *Hermitian* if it is equal to its conjugate transpose $(H = H^*)$. In other words, matrix H is Hermitian if

$$\overline{H_{i,j}} = H_{j,i}, \quad \forall i = 1, \ldots, d, j = 1, \ldots, d.$$

It can be shown that the eigenvalues of these kind of matrices are always real numbers. That means that we have an ordering of the eigenvalues of a Hermitian matrix. Additionally, there exists an orthonormal basis $u_1, u_2, ..., u_d$ for $\mathbb{C}^d$ made up of the eigenvectors of H. So the eigenspaces of a hermitian matrix are mutually orthogonal. Actually, if we put vectors u_i as columns of the matrix U than there exists a diagonal matrix Λ such that

$$A = U\Lambda U^*. \tag{2.2}$$

All of these properties make solving the Hermitian eigenproblem easier than solving general eigenproblem. The eigensolvers are usually classified by the approach which is used for calculating the eigenpairs (λ, x). *Direct eigensolvers* perform transformations on a matrix in order to diagonalize it. These methods are performing similarity transformations which should bring matrix H to its diagonal form, from which we can easily read the eigenvalues. While direct eigensolvers are usually used on dense matrices, for sparse matrices we use *iterative solvers*. In contrast to direct eigensolvers, iterative solvers perform transformations on vectors. The idea is to take one or more starting vectors, iteratively perform transformations until they converge to the eigenvectors. These sorts of methods are used mostly on the sparse matrices due to cheap execution of transformations. We will now focus on the iterative methods. These methods are more flexible when calculating only a small part of the spectrum. Also, if starting vector is already a good approximation for an eigenvector then it will take a small amount of work to calculate the eigenvector. These assumptions, together with the polynomial acceleration strategy, present a good argument for using iterative methods to solve dense eigenvalue problems.

2.1 Subspace iterations and ChASE

The power method is one of the simplest iterative methods for extracting a single eigenpair of a matrix. The method consists of generating a sequence of vectors $H^k v_0$, where v_0 is some nonzero initial vector. This method, when converges, calculates the dominant eigenvector, which is the eigenvector associated with the eigenvalue with the largest modulus. Since there is an orthonormal basis $u_1, ..., u_d$ of eigenvectors of H, an initial vector v_0 can be written as a linear combination of eigenvectors.

$$v_0 = \sum_{i=1}^{d} \gamma_i u_i. \tag{2.3}$$

By taking H and multiplying it with v_0, taking into the account the linearity of multiplication, we get:

$$Hv_0 = \sum_{i=1}^{d} \gamma_i \lambda_i u_i.$$

So the k-th iteration looks like this:

$$H^k v_0 = \sum_{i=1}^{d} \gamma_i \lambda_i^k u_i.$$

If we assume that the eigenvalues are given in the descending order of their mudulus, $|\lambda_1| > |\lambda_2| \geq |\lambda_2| \geq ... \geq |\lambda_d|$, then we could extract the most dominant eigenvalue out of

the sum

$$H^k v_0 = \lambda_1^k [\gamma_1 u_1 + \sum_{i=2}^{d} \gamma_i (\frac{\lambda_i}{\lambda_1})^k u_i].$$

By repeatedly normalising this sequence we get that all the components, except for the most dominant one, tend to zero, since $(\frac{\lambda_i}{\lambda_1})^k \to 0$ when $k \to \infty$.

This approach can be easily generalized in the sense that we can calculate several most dominant eigenvectors at once. Starting with an initial system of *nev* vectors forming a $d \times nev$ matrix $X_0 = [x_1, ..., x_{nev}]$, we proceed with the similar approach as with the power method. The iterations consist of computing the matrix $X_k := H^k X_0$. As we have already stated, all of the vector columns of X would converge to the most dominant eigenvector. This means that the column vectors in X_k will progressively loose their linear independence. For that reason we reestablish their linear independence by reorthonormalizing them again after every few iterations. This method is called Subspace Iterations. It is capable of calculating m most dominant eigenvectors of a matrix.

However, in either case, the rate of convergence of these methods could be quite slow. The rate of convergence greatly depends on the ratios $|\frac{\lambda_i}{\lambda_1}|$ which could be quite close to 1. In order to improve the rate of convergence one could use the *polynomial acceleration*. ChASE eigensolver utilizes this strategy in order to both accelerate the method and to make filtration of eigenvalues.

Sometimes, it is of interest to calculate only a smaller part of the spectrum. For this, one should filter out the unwanted eigenvalues. The way that ChASE is doing this is by using the Chebyshev filtering.

2.2 Chebyshev filter

The way in which the subspace iterations perform is to successively calculate the product $H^k X_0$. This approach corresponds to taking the polynomial $p(t) = t^k$ and calculating $p(H)X_0$. The idea for improvement is to use more general polynomial which could improve the rate of convergence. By taking the general polynomial p in the power method, we get:

$$p(H)v_0 = \sum_{i=1}^{d} \gamma_i p(\lambda_i) u_i.$$

If we choose the polynomial such that the value $p(\lambda_i)$ is large for some $i = 1, 2, ..., d$ and small for the others, then we would get the convergence towards this chosen i. In this way, we filter out unwanted eigenvalues and accelerate the convergence towards wanted ones. One might wonder which polynomial would give the fastest convergence?

The answer comes in the form of the Chebyshev polynomials. These polynomials could be used in order to neglect the eigenvalues in a certain interval and amplify the convergence to the rest. The Chebyshev polynomials are defined with the three-term recurrence relation

$$C_{k+1}t = 2tC_k(t) - C_{k-1}(t), \quad C_0(t) = 1, C_1(t) = t. \tag{2.4}$$

The relation (2.4) allows us to evaluate the polynomials by iterating it up to the desired degree. The following is an important result from the approximation theory which

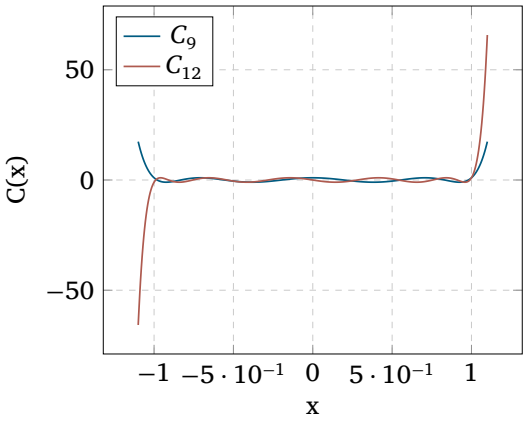


Figure 2.1: Examples of Chebyshev polynomials

lightens up the importance of the Chebyshev polynomials. Let $[\alpha, \beta]$ be a non-empty interval in $\mathbb{R}$ and let $\gamma \geq \beta$ be a scalar. Then the solution to the optimization problem

$$\min_{p \in \mathbb{P}_k, p(\gamma)=1} \max_{t \in [\alpha,\beta]} |p(t)| \tag{2.5}$$

is given with the scaled and shifted Chebyshev polynomial

$$\hat{C}_k(t) := \frac{C_k(1 + 2\frac{t-\beta}{\beta-\alpha})}{C_k(1 + 2\frac{\gamma-\beta}{\beta-\alpha})}. \tag{2.6}$$

By using the notation $c := \frac{\beta+\alpha}{2}$, $e := \frac{\beta-\alpha}{2}$ we can rewrite the polynomial (2.6) as $\hat{C}_k(t) = \frac{C_k((t-c)/e)}{C_k((\gamma-c)/e)}$. Taking into account the relation (2.4), we can simplify the calculation of polynomials $\hat{C}_k$. Letting $\rho_k := C_k((\gamma - c)/e)$, $k = 1, 2, ...$, we get:

$$\rho_{k+1}\hat{C}_{k+1}(t) = C_{k+1}(\frac{t-c}{e}) = 2\frac{t-c}{e}C_k(\frac{t-c}{e}) - C_{k-1}(\frac{t-c}{e}) = 2\frac{t-c}{e}\rho_k\hat{C}_k(t) - \rho_{k-1}\hat{C}_{k-1}(t).$$

By defining $\sigma_{k+1} = \rho_k/\rho_{k+1}$, we get the relation:

$$\hat{C}_{k+1}(t) = 2\sigma_{k+1}\frac{t-c}{e}\hat{C}_k(t) - \sigma_k\sigma_{k-1}\hat{C}_{k-1}(t).$$

We are able to calculate the recurrence relation for σ_i, $i = 1, 2, ...$:

$$\begin{aligned} \sigma_1 &= \frac{e}{\gamma - c}; \\ \sigma_{k+1} &= \frac{1}{\frac{2}{\sigma_1} - \sigma_k}, k = 1, 2, ... \end{aligned}$$

Now we can derive the relation for the Chebyshev iterations. Starting with the initial vector v_0, we have:

$$v_1 = \frac{\sigma}{e}(H - cI)v_0, \tag{2.7}$$

$$v_{k+1} = \hat{C}_{k+1}(H)v_0 = 2\frac{\sigma_{k+1}}{e}(H - cI)\hat{C}_k(H)v_0 - \sigma_k\sigma_{k+1}\hat{C}_{k-1}(H)v_0 \tag{2.8}$$

$$= 2\frac{\sigma_{k+1}}{e}(H - cI)v_k - \sigma_k\sigma_{k+1}v_{k-1}. \tag{2.9}$$

The same holds for the case with multiple vectors. We will denote the shifted matrix $H - cI$ as $\hat{H}$. Note that the matrix $\hat{H}$ remains Hermitian.

Since all the eigenvalues of the Hermitian matrices reside on a real line, by using the Chebyshev polynomials, we are capable of filtering out the eigenvalues in an interval of our choice. In practice, we would first calculate the estimate for the most dominant eigenvalue λ_1 and set $\gamma = |lambda_1|$. We define the interval $[\alpha, \beta]$ in a way that α is mapped to a lower estimate for $|\lambda_d|$ and β is mapped to the estimate of $|\lambda_nev|$. In this way, we get the improved convergence towards *nev* most dominant eigenvalues. Using the three-term recurrence relation (2.9) we can calculate iterations for the Chebyshev filter. A lot more about eigensolvers can be found in [5].

Chebyshev filter takes up the most of the computation time of the whole solver, as we can see on the Figure 2.2. This gives us a good motivation for introducing parallel programming strategies.

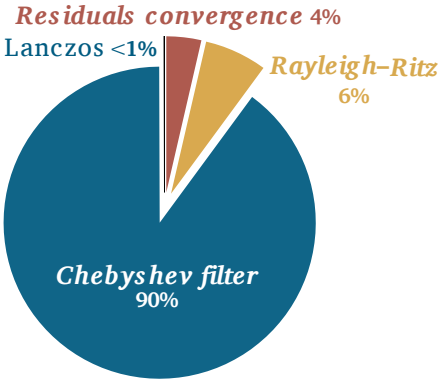


Figure 2.2: Time usage chart for ChASE

More on the ChASE eigensolver and applications can be found in [1], [4] .

3 Strategies of parallelization

In this section we describe the strategies which we use for implementing the Chebyshev polynomial acceleration. This algorithm consists of successive iteration of the three terms recurrence relation of the form

$$X^{(m+1)} = \alpha_m\hat{H}X^{(m)} - \beta_m X^{(m-1)}, \quad m = 2, 3, ..., k \quad X^{(1)} = \hat{H}X^{(0)} \quad \hat{H} \in \mathbb{C}^{d\times d}, X^{(i)} \in \mathbb{C}^{d\times nev}, \tag{3.1}$$

where parameters β_m and α_m are determined with the scaling and shifting of the Chebyshev polynomials. The recursive nature of this formula allows us to evaluate it using only 3 arrays, A, B and C. Indeed, we use the array A to store the matrix $\hat{H}$ and the array B to store the initial matrix $X^{(0)}$. The first iteration is done by calculating

$$C \leftarrow \alpha_0 AB. \tag{3.2}$$

The second iteration utilizes the array B for saving the product:

$$B \leftarrow \alpha_1 AC - \beta_1 B. \tag{3.3}$$

The third iteration shifts back to C,

$$C \leftarrow \alpha_2 AB - \beta_2 C, \tag{3.4}$$

and the process goes on until we reach the desired degree of a Chebyshev polynomial.

The largest part of these iterations is the matrix-matrix multiplication. In order to efficiently perform the multiplication we exploit graphics processing units (GPUs) since their architecture is favourable for such linear algebra operations. On the other hand, GPUs have much less memory then required for saving the whole matrix H. Such matrices, which we get from applications, are about the size $10^5 \times 10^5$ which corresponds to 160 GBs of memory. This means that implementing the multiplication using GPUs will require the use of multiple GPU devices at once. In practice we will divide the calculation over multiple GPU nodes which contain several GPUs each.

Managing such distribution requires multiple processes together with an appropriate communication layout. In the next section, we describe the scheme for performing one Chebyshev filter iteration.

3.1 Distribution among nodes, employment of GPUs

We first have to tile up the matrix A and distribute the tiles on different processes. Since

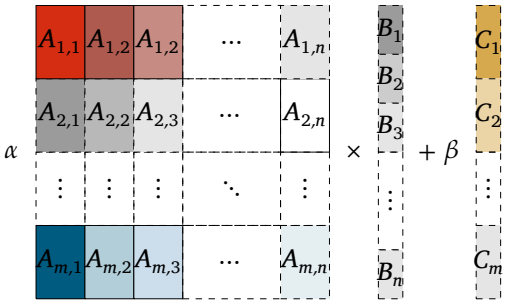


Figure 3.1: Tiling of matrices A, B and C

we are dealing with the tiling of a matrix, we will use Cartesian grid of processes which fits perfectly with the usual matrix tiling scheme. Each node will perform one process and receive one tile of a matrix A together with the appropriate tiles of matrices B and C. It is necessary to mention that, together with the arrays A, B and C, each node also has

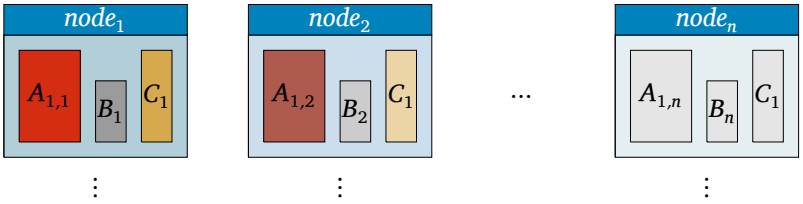


Figure 3.2: Array distribution on nodes

the working array *IMT* which has an important role as an intermediate array between calculating AB and $\alpha AB - \beta C$.

The first step in calculating the iteration is performing the multiplication AB. For the sake of consistency we assume that we have 4 GPU devices available on each node, however usually we will use all the GPUs at our disposal. The next step is distributing

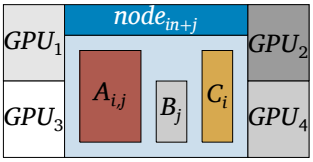


Figure 3.3: Node structure

the data on GPUs in order to perform multiplication. There are several ways to do it. From now on, whenever we focus on one node, we will be using the notation A, B and C instead of $A_{i,j}$, B_j and C_i, for the sake of simplicity.

We begin with the vertical or VER implementation, in which we divide A in vertical blocks as following:

$$A = \begin{bmatrix} A_1 & A_2 & A_3 & A_4 \end{bmatrix}, \qquad B = \begin{bmatrix} B_1 \\ B_2 \\ B_3 \\ B_4 \end{bmatrix}. \tag{3.5}$$

In this way, the calculation of the product AB reduces to the calculation of smaller products A_iB_i since

$$AB = \sum_{i=1}^{4} A_iB_i.$$

Each of the smaller products can be calculated concurrently on separate GPU and then summed into the final product.

The next implementation is the horizontal or HOR implementation. There, we also divide A into 4 blocks but this time horizontally. We have:

$$A = \begin{bmatrix} A_1 \\ A_2 \\ A_3 \\ A_4 \end{bmatrix},$$

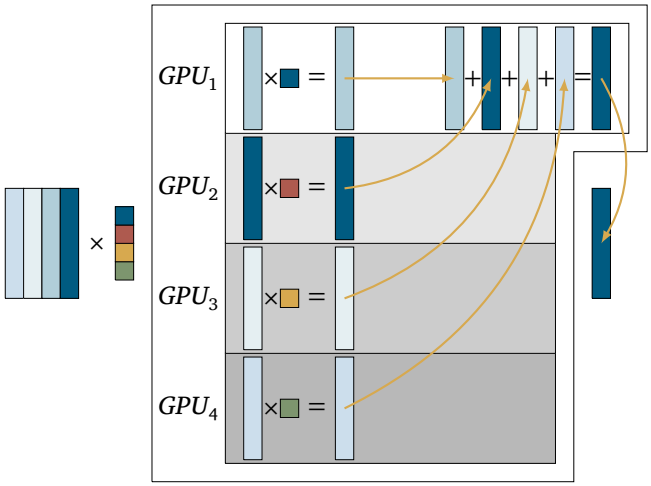


Figure 3.4: Vertical scheme

so we have to calculate 4 products $A_iB = C_i$ and then concatenate C_i to get

$$C = \begin{bmatrix} C_1 \\ C_2 \\ C_3 \\ C_4 \end{bmatrix}.$$

The third implementation is the mixed or HV implementation. This time we divide A

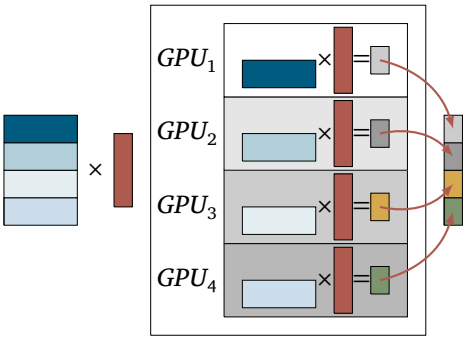


Figure 3.5: Horizontal scheme

into 4 blocks as follows:

$$A = \begin{bmatrix} A_1 & A_2 \\ A_3 & A_4 \end{bmatrix}, \quad B = \begin{bmatrix} B_1 & B_2 \end{bmatrix}.$$

We have:

$$C = \begin{bmatrix} C_1 \\ C_2 \end{bmatrix} = \begin{bmatrix} A_1B_1 + A_2B_2 \\ A_3B_1 + A_4B_2 \end{bmatrix}.$$

The way this is calculated is to first distribute the data, then calculate the products on separate GPUs. Then we take i.e. the product A_2B_2 and send it to the GPU containing A_1B_1 so that they can be summed. Afterwards we concatenate the results. Upon implementing

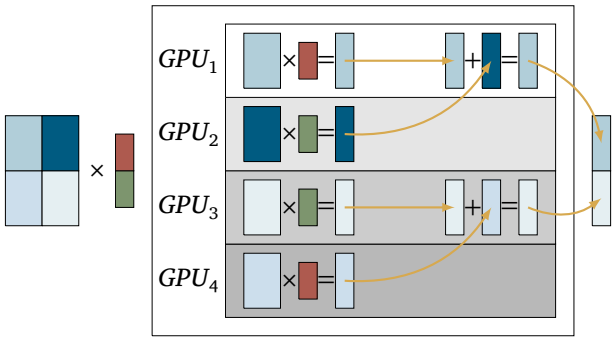


Figure 3.6: Mixed scheme

these schemes, we performed experiments in order to decide which one to use as the default one. Experiments were performed with different matrix sizes and shapes. Upon

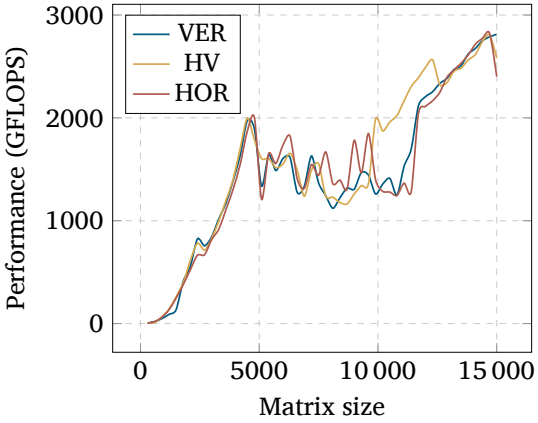


Figure 3.7: Performance comparison plot

examining the results in the figure 3.7, we opted for the HV implementation whenever we have 4 GPU devices available on a node. Otherwise we will use either VER or HOR implementation.

We will discuss the actual implementation issues in the next section. Lets just emphasize that at the end of the multiplication on each node, the array *IMT* contains the product.

3.2 Alternating cycle algorithm for evaluating Chebyshev filter

So far we are able to calculate the product *AB* on each node using the available GPUs.

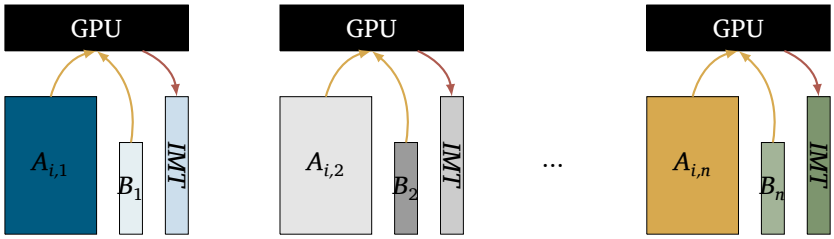


Figure 3.8: Step 1

In order to perform full Chebyshev iteration we must first develop a communication strategy between processes. Since the matrix A was tiled up, we will get the final product by summing the contributions from each process in the same row.

$$AB = \begin{bmatrix} A_{1,1} & A_{1,2} & \dots & A_{1,n} \\ A_{2,1} & A_{2,2} & \dots & A_{2,n} \\ \vdots & \vdots & \vdots & \vdots \\ A_{m,1} & A_{m,2} & \dots & A_{m,n} \end{bmatrix} \times \begin{bmatrix} B_1 \\ B_2 \\ \vdots \\ B_n \end{bmatrix} = \begin{bmatrix} \sum_{j=1}^{n} A_{1,j}B_j \\ \sum_{j=1}^{n} A_{2,j}B_j \\ \vdots \\ \sum_{j=1}^{n} A_{m,j}B_j \end{bmatrix}.$$

In order to get the correct products on each node, we perform one reduction (summing) of arrays IMT on each row of processes. After this, we will in each row have the tile of the correct result AB saved in the array IMT.

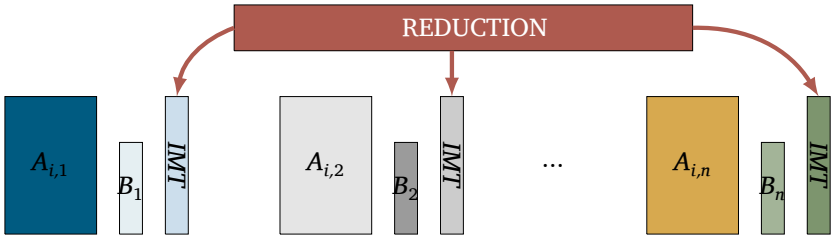


Figure 3.9: Step 2

Upon finishing the reduction, it is straight forward to calculate the final Chebyshev iteration $C \leftarrow \alpha IMT - \beta C$ by using BLAS routines scal and axpy. After this step, each process possesses one tile of the matrix $\alpha\hat{H}X^{(k)} - \beta X^{(k-1)}$ which corresponds to the row of processes which it belongs to. This tile is saved in the array C.

Now that we have finished one cycle of the Chebyshev filter, we have to continue with the calculation of $B \leftarrow \alpha AC - \beta B$. Here, we face more problems with the implementation. At this point, the processes in each row of the Cartesian grid have the same tile of the array C. For almost every process this tile of C is utterly useless since it is not the tile which should take part in the next iteration. In adition, also the blocks of C are not compatible with the tiling of the matrix A. That means that if we want to successfully multiply A with C we have to redistribute parts of C to the according processes, together with the slicing and glueing of different tiles. However, there is a way to avoid this complicated communication. The matrices from the problems which are of interest

to us are Hermitian. This, seemingly unrelated fact will enable us to elegantly solve communication issues and implement fairly efficient scheme.

The most important thing is to remember that the Hermitian matrices are the same as their transpose conjugates. We can exploit this fact in a very straight forward way. If we would take the transpose conjugate of matrix A then we would also take the transpose conjugate of its tiles. The tiling of the matrix C would then match with the tiling of the conjugate transpose of A. Also, another convenient thing would happen. The tiles of the matrix C would suddenly appear on all the right processes where they are needed.

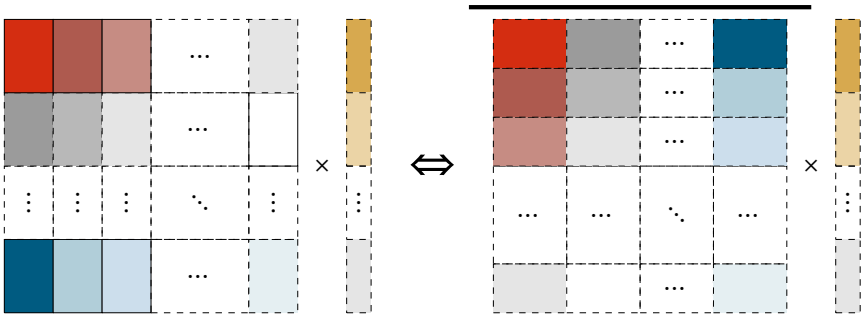
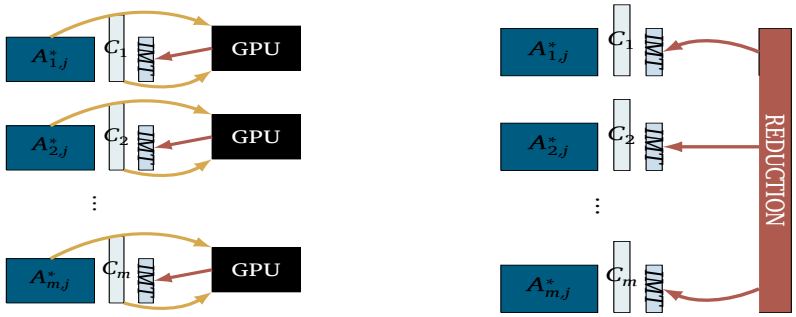


Figure 3.10: The conjugate transpose equivalence

However, we don't actually transpose the matrix A. What we do is virtually transpose the Cartesian grid, then multiply the conjugate transposes of tiles of A with the tiles of C. Actually, the tiles of C were in all the right places this whole time.

What were once rows now are columns. Like before, we perform A^*C multiplication on GPUs and retrieve the product back to the CPU, then save it inside IMT. Afterwards we have to perform the reduction to sum up all the parts of the product in each column of processes. We can take a look at the Figure 3.11 to examine the scheme. The practical



(a) Step 1 (b) Step 2

Figure 3.11: Second cycle of Chebyshev filter

implementation only calls different types of the multiplication routines (A^*C or AB) and accordingly performs a reduction on either rows or columns. Notice that, in this way, we have reduced the communication between processes to performing only one reduction

per cycle. This way of iterating the recurrence relation (3.1) is quite efficient and scales well.

There is one more detail which we have to discuss. In order to avoid unnecessary data copying to GPUs, we distribute the matrix A only once, at the beginning of the Chebyshev filter. This means that the tiles of matrix A, once distributed on GPUs, remain there in the same exact form for the duration of the Chebyshev filter. However, since we sometimes have to transpose the matrix, we will have to alternate between different schemes for the matrix-matrix multiplication. For example, in the case of 3 available GPU devices per node, we would have to alternate between VER and HOR implementations.

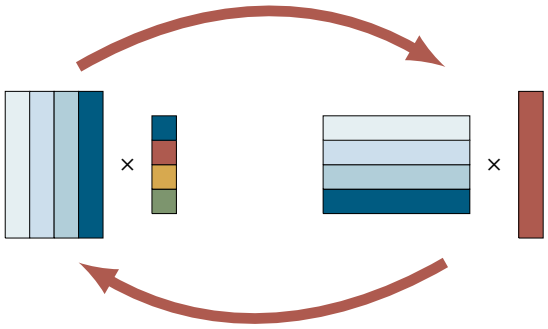


Figure 3.12: Alternating usage of schemes

In the next section we discuss the actual implementation in C/CUDA/MPI together with examining the results of the scaling experiments.

4 Implementation

For the actual implementation we used the programming languages C and C++. The code is divided into two parts. The first part is related to the multi GPU programming while the other is related to managing multiple processes on multiple nodes.

4.1 Multi GPU matrix-matrix multiplication

The first part that we implemented is multi GPU matrix-matrix multiplication on one node. For programming GPUs we used CUDA. CUDA is parallel computing platform which enables us to use CUDA-enabled GPUs for general purpose processing.

The center of this program is the structure called GPU_Handler.

Listing 4.1: GPU_Handler structure

```
1
2 typedef struct _GPU_Handler {
3          cuDoubleComplex **A_dev;
4          cuDoubleComplex **B_dev;    //arrays on GPUs
5          cuDoubleComplex **C_dev;
6          cuDoubleComplex **TMP;      //special array for temorary results
7          cublasHandle_t *handle;     //array of device handles
8          cudaStream_t *stream;       //array of cudaStreams
9          int m;                      //vertical dimension of A
10         int n;                      //horizontal dimension of A
11         int l;                      //horizontal dimension of B
```

```
12          int NOD;                    //number of devices
13          int *k;
14          int *off;                   //arrays of block dimensions
15          bool use_2D;                //this tells us which scheme to use
16  } GPU_Handler;
```

Its purpose is to keep track of all the allocated memory and parameters which are used throughout the execution of the program. Cuda provides us structure called cuDoubleComplex which implements double precision complex number. We will use that structure to implement matrix entries. In order to achieve parallelism, we use different cudaStream objects for each GPU. In order to perform the concurrent memory transfer we also have to use the pinned memory for arrays on the CPU.

Here we faced the first coding obstacle. For the usual transfer of data arrays using multiple streams, one could use the function cudaMemcpyasync(). The problem is that we wish to copy chunks of arrays (tiles) which do not make a contiguous memory array. However, we can overcome this obstacle by using the function cudaMemcpy2DAsync(). In order to successfully use cudaMemcpy2DAsync(), one should provide the host pitch, the device pitch, height, width and a pointer to the first element in the block.

LISTING 4.2: MEMCPY2DASYNC USED IN HV SCHEME FOR TRANSFERING TILES OF A

```
1 pitch_host = GPU_hand->m*sizeof(cuDoubleComplex);
2           for(i=0;i<GPU_hand->NOD;i++)
3           {
4                   pitch_device = ky[i/2]*sizeof(cuDoubleComplex);
5                   cuda_exec(cudaMemcpy2DAsync(GPU_hand->A_dev[i], pitch_device, A+kx[0]*GPU_hand->m*(
                        i%2)+ky[0]*(i>1), pitch_host,     ky[i/2]*sizeof(cuDoubleComplex), kx[i%2],
                        cudaMemcpyHostToDevice, GPU_hand->stream[i]));
6           }
```

The pitch is an argument which holds the information on how far in the memory is one array column from the other.

Now that we have concurrent memory transfer covered, we move on to the multiplication. CUDA provides cuBLAS library, which implements basic linear algebra operations. We use cublasZgemm() for the multiplication of double precision complex matrices and cublasZaxpy() for adding them.

LISTING 4.3: ZGEMM USED IN HV SCHEME FOR MULTIPLYING TILES OF A AND B

```
1 for(i=0;i<GPU_hand->NOD;i++)
2           {
3                   cuda_exec(cudaSetDevice(i));
4                   cublasZgemm(GPU_hand->handle[i], CUBLAS_OP_N, CUBLAS_OP_N, ky[i/2], l, kx[i%2], &a,
                         GPU_hand->A_dev[i], ky[i/2],     GPU_hand->B_dev[i], kx[i%2], &b, GPU_hand->
                        C_dev[i], ky[i/2]);
5           }
```

All the code which we use to control GPUs is distributed onto several functions in the MMMGPU library. While working on the host CPU we use MKL_Complex16 implementation of double complex numbers. This requires casting the array pointers into the right type before calling an appropriate GPU function.

Next, we implemented the library for managing multiple processes on different nodes and calculating Chebyshev iterations.

4.2 MULTI CPU CHEBYSHEV ITERATIONS

For running multiple processes and programming the communication we used Massage Passing Interface – MPI. MPI is a standardized message-passing system which one can

LISTING 4.4: MMMGPU LIBRARY

```
 void GPU_init_1D(GPU_Handler *GPU_hand);
 void GPU_init_2D(GPU_Handler *GPU_hand);
 void GPU_init(int m, int n, int l, GPU_Handler *GPU_hand); //calculates parameters and allocates
      memory

void GPU_load_2D(cuDoubleComplex *A, GPU_Handler *GPU_hand);
void GPU_load_1D(cuDoubleComplex *A, GPU_Handler *GPU_hand);
void GPU_load(MKL_Complex16 *A, GPU_Handler *GPU_hand);     //distributes tiles of A

void GPU_doGemm_2D_AB(cuDoubleComplex *B, cuDoubleComplex *C, int l, GPU_Handler *GPU_hand);
void GPU_doGemm_2D_AC(cuDoubleComplex *B, cuDoubleComplex *C, int l, GPU_Handler *GPU_hand);
void GPU_doGemm_1D_AB(cuDoubleComplex *B, cuDoubleComplex *C, int l, GPU_Handler *GPU_hand);
void GPU_doGemm_1D_AC(cuDoubleComplex *B, cuDoubleComplex *C, int l, GPU_Handler *GPU_hand);
void GPU_doGemm(MKL_Complex16 *B, MKL_Complex16 *C, int l, GPU_Handler *GPU_hand, char mode); //
     performs multiplication and returns the result

void GPU_destroy_1D(GPU_Handler *GPU_hand);
void GPU_destroy_2D(GPU_Handler *GPU_hand);
void GPU_destroy(GPU_Handler *GPU_hand);        //frees memory
```

use for parallel programming. Again, the center of the code is the structure called CPU_Handler. Each process will initialize its own CPU_Handler which will keep track of all the arrays and communicators which it will use. CPU_Handler also possesses one instance of GPU_Handler so that it can use GPUs available on its node.

LISTING 4.5: CPU_HANDLER

```
typedef struct _CPU_Handler {
         MPI_Group ROW, COL, origGroup; //groups of processes (rows and columns groups)
         MPI_Comm ROW_COMM, COL_COMM;   //each process needs one row communicator and one column
              communicator
         MPI_Comm CART_COMM;    //cartesian communicator
         int *ranks_row;
         int *ranks_col;
         int dims[2];   //optimal dimensions of cartesian grid
         int coord[2];
         int off[2];
         MKL_Complex16 *A, *B, *C;  //tiles of matrix H, X
         MKL_Complex16 *IMT;    //intermediate array
         int global_n, m, n, nev, nprocs, rank;
         char next;     //keeps track of multiplication cycle ('B' or 'C')
         int initialized;
         GPU_Handler GPU_hand;  //handler for managing GPUs on this process/node
 } CPU_Handler;
```

There is an especially useful function for calculating grid dimensions for our 2D Cartesian grid scheme. It is called MPI_Dims_Create(). For a given number of processes it calculates the most square Cartesian grid possible.

LISTING 4.6: CREATING CARTESIAN COMMUNICATOR

```
    MPI_Dims_create(CPU_hand->nprocs, 2, CPU_hand->dims);
    MPI_Cart_create(MPI_COMM_WORLD, 2, CPU_hand->dims, periodic, reorder, &(CPU_hand->CART_COMM));
```

The Cartesian communicator offers useful translation between usual process ranks and Cartesian coordinates in the form of MPI_Cart_coords() and MPI_Cart_rank() functions.

LISTING 4.7: TRANSLATION

```
MPI_Cart_coords(CPU_hand->CART_COMM,CPU_hand->rank,2,CPU_hand->coord); //computes coordinates from
     rank
MPI_Cart_rank(CPU_hand->CART_COMM,coord,&r);    //computes rank from coordinates
```

In order to perform reductions we must create communicators for each row and column of processes. Since each process belongs to exactly one row and one column, each process needs only one row communicator and one column communicator. We do it by creating groups of processes and then creating according communicators.

LISTING 4.8: CREATING ROW AND COLUMN COMMUNICATORS

```
1 MPI_Comm_group(CPU_hand->CART_COMM, &(CPU_hand->origGroup)); //creates original group of processes
2 MPI_Group_incl(CPU_hand->origGroup, CPU_hand->dims[1], CPU_hand->ranks_row, &(CPU_hand->ROW)); //
      creates group of all the processes in this row
3 MPI_Group_incl(CPU_hand->origGroup, CPU_hand->dims[0], CPU_hand->ranks_col, &(CPU_hand->COL)); //
      creates group of all the processes in this column
4 MPI_Comm_create(MPI_COMM_WORLD, CPU_hand->ROW, &(CPU_hand->ROW_COMM));
5 MPI_Comm_create(MPI_COMM_WORLD, CPU_hand->COL, &(CPU_hand->COL_COMM));  //create required
      communicators
```

One Chebyshev iteration consists of the next steps:

LISTING 4.9: ONE ITERATION OF CHEBYSHEV FILTER

```
1 GPU_doGemm(CPU_hand->B,CPU_hand->IMT,CPU_hand->nev,&(CPU_hand->GPU_hand),CPU_hand->next);  //perform
       multiplication
2 MPI_Allreduce(MPI_IN_PLACE, CPU_hand->IMT, CPU_hand->m*CPU_hand->nev, MPI_DOUBLE_COMPLEX, MPI_SUM,
      CPU_hand->ROW_COMM);  //reduce
3 zscal(&dim,&beta,CPU_hand->C,&inc);
4 zaxpy(&dim,&alpha,CPU_hand->IMT,&inc,CPU_hand->C,&inc);  //calculate alphaAB+betaC
5 CPU_hand->next = 'C';   //reminder that the next iteration is  alphaAC+betaB
```

Now we have everything ready. First we distribute the data. Then we do the iteration multiple times, return the result and free the memory. The code is organised as a library called CHEBMPI.

LISTING 4.10: CHEBMPI LIBRARY

```
1 void CPU_handler_init(CPU_Handler *CPU_CPU_hand, int global_n, int nev);    //initialize everything
2 void CPU_load(CPU_Handler *CPU_CPU_hand);   //load tiles of H, X
3 void CPU_doCheb(CPU_Handler *CPU_hand, MKL_Complex16 alpha, MKL_Complex16 beta);    //perform
      iteration
4 void CPU_destroy(CPU_Handler *CPU_hand);    //free memory
5 void CPU_get_off(CPU_Handler *CPU_hand, int *xoff, int *yoff, int *xlen, int *ylen);    //calculate
      offsets in regard to matrix H
6 void CPU_get_C(CPU_Handler *CPU_hand, int *COff, int *CLen, MKL_Complex16 *C);      //acquire the
      filtered matrix
```

Once the filtering is complete, one calls function CPU_get_C() on each process which then returns the calculated tile of C. More about CUDA and MPI programming can be found in [2], [3].

5 RESULTS

We have performed experiments with this scheme in order to get a good idea of how it behaves in different situations.

We have measured times of execution for a different number of processes. The size of the matrix which we distribute is $10^5 \times 10^5$, the number of target eigenvectors (nev) is equal to 1% of the size of the matrix. We ran 10 cycles of the Chebyshev iterations for several different random matrices and calculated the average times of execution. In the Figure 5.1 we see the average times of execution for one Chebyshev iteration in regards to the number of nodes involved. This sort of experiment is called *strong scaling*. The strong scaling plot indicates that the program scales well up to 15 processes. Then the scaling stops, and the speed gained with the multiplication is undid with the time spent

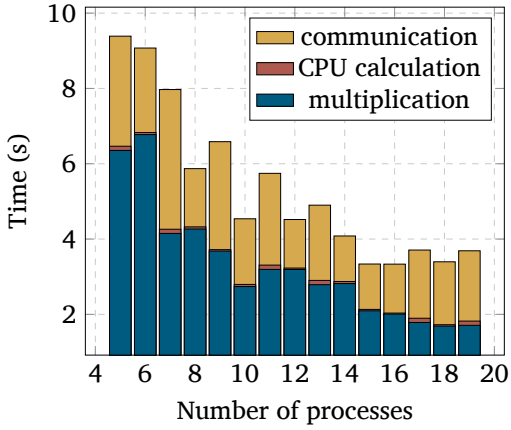


Figure 5.1: Strong scaling plot, $d = 10^5$

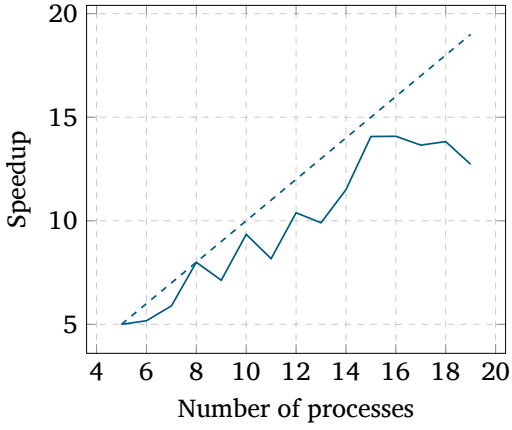


Figure 5.2: Speed-up, $d = 10^5$

Figure 5.3: Weak scaling plot, $d = 40\,000 \times \sqrt{\#nodes}$

on communication. From the time measured, the speed-up can be calculated. Figure 5.2 also shows how the speed-up is lost at some point.

We also did the *weak scaling* experiment. There we maintained constant problem size per node. It is done by increasing the matrix size by multiplying it with the square root of the number of processes. The weak scaling plot indicates that we should avoid using prime number of nodes. Using prime number of nodes results in one dimensional Cartesian grid of processes which yields thin rectangular tiles. This prolongs communication and time spend doing the matrix-matrix multiplication.

All in all, the scaling results are not bad, the performance is quite fast. There is surely a lot to do in order to improve the scheme, but we can already draw some conclusions. It is best to use a number of processes which yields the most square Cartesian grid. That will make the communication during cycles faster and more uniform. It will also make matrix tiles more square which will result in better performance of the matrix-matrix multiplication.

6 Outlook

Further research should be directed into finding out what causes problems with scaling for larger number of processes. That requires better insight into the scheme and the usage of some sort of profiler. We are working on calculating execution times in more details in order to detect potential bottlenecks. One of the next steps is developing general 2D scheme for the GPU multiplication similar to HV. This scheme would recognize the number of available GPU devices and accordingly perform multiplication using a Cartesian grid, not only for 4 devices, but also for 6,8... Further performance experiments should be performed once this scheme is integrated with ChASE. We expect this to reduce the amount of time spent during Chebyshev filter. It is also necessary to take a closer look at some of the other parts of the ChASE eigensolver and see what can be done to parallelize them. This will lead to an optimized program which utilizes several parallel programming aspects in order to increase its efficiency.

7 Acknowledgements

I would like to express my gratitude to my supervisor Dr. Edoardo Di Napoli for guiding me through this programme. Together with Jan Winkelmann, he gave me valuable advices and remarks that were incredibly useful for my project. I still haven't seen any of the terrible weather that Edoardo was warning me about, though.

Special thanks to Dr. Ivo Kabadshow for making all of this happen. His constant support made this programme very enjoyable and pleasant.

I want to thank all of the guest students for all the great times I've spent with them. I wish them all good fortune with their future life and career.

References

[1] M. Berljafa and E. D. Napoli. **A parallel and scalable iterative solver for sequences of dense eigenproblems arising in FLAPW**. *CoRR*, abs/1305.5120, 2013. http://arxiv.org/abs/1305.5120.

[2] N. Corporation. *NVIDIA CUDA Compute Unified Device Architecture Programming Guide*. NVIDIA Corporation, 2007.

[3] T. M. Forum. **Mpi: A message passing interface**, 1993.

[4] E. D. Napoli. **Numerical methods for the eigenvalue problem in electronic structure computations**. 2014.

[5] Y. Saad. *Numerical methods for large eigenvalue problems*, volume 158. SIAM, 1992.

Large Eddy Simulations for Turbulence

Modeling and Implementation

Suryanarayana Maddu
Department of Environment and Civil Engineering
Ruhr University Bochum
Germany
Suryanarayana.MadduKondaiah@ruhr-uni-bochum.de

Abstract *Turbulence modelling is one of the most important physical phenomenon for flow and heat transport modeling. Accurate modeling and implementation of the physics is quintessential for comprehensive system analysis. In this direction, we investigate Large Eddy simulation for turbulent modeling for flow in simple geometries for preliminary analysis. Two common models have been implemented: Constant Smagorinsky and Dynamic Smagorinsky. Qualitative benchmarking for the diffusion problem is conducted and the algorithm is ported to GPU using OpenACC.*

1 Introduction

Smoke propagation through complex geometries requires modeling of fluid flow, heat transport and turbulence coupled with complex physics of combustion, radiation and soot formation. It is a tough numerical challenge because it requires the solution of continuum equations for fluid flow, heat transfer and turbulence a shown in Figure 1.1. These physics can than be coupled to a fire model that incorporates the effects of combustion, radiation and soot formation. Pyrolysis, forced convection, radiation and air entrainment can also be included into the model (Figure 1.2). The first steps to solve this problem is to solve the Navier-Stokes and the energy equations for fluid flow and heat transport numerically.

The proposed continuum equations for fluid flow, heat and turbulence models are to be discretized and solved efficiently. So, the finite difference method is chosen as the discretization technique because of its ease of implementation and solid mathematical basis. Another reason for choosing finite difference is its implicit functionality as filter for attenuation of high frequencies used in turbulence modeling.

Smoke flow is usually turbulent because of the very low density and viscosity of the smoke. So it is important to model and include the turbulent effects with good trade-off between efficiency and accuracy into the real-time simulation for smoke propagation. For this purpose, we implemented the Large Eddy Simulation (LES) based on Sub-grid Scale (SGS) model where the large scale fluid motion is computed and the small scale motion is modeled. The two most common Sub-grid Scale models are Constant Smagorinsky and Dynamic Smagorinsky. Real-time simulation for large systems and complex geometries requires the most efficient and scalable code for instant feedback and steering of the simulation parameters. Since most of the discretization is based on the local stencil finite difference operations, it can be ported for multi-core and GPU implementation effectively. We use OpenACC for code acceleration due to its simple programming syntax and minimal changes to the code. Using simple *pragma* constructs before parallelizable loops, we can achieve significant speedup of the serial code.

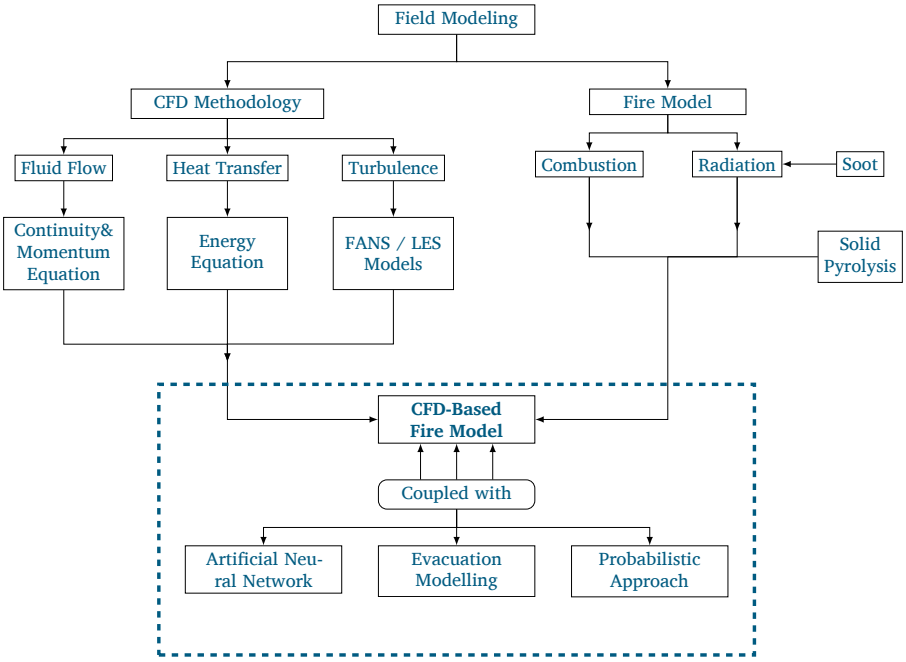


Figure 1.1: Overview of modeling steps for smoke simulation [14].

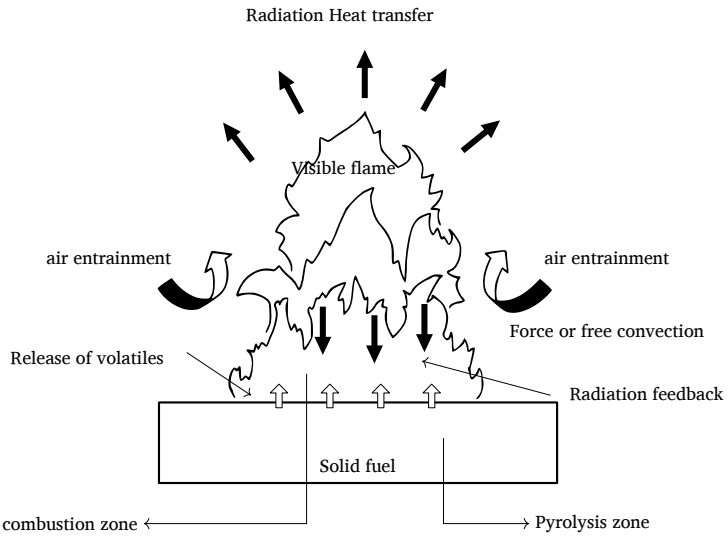


Figure 1.2: Processes involved in the smoke generation and propagation [14].

2 Governing equations and discretization

2.1 Equations of motion

The most notable equations in fluid dynamics are the Navier-Stokes equations(NSE) [1]. Anything which has fluid characteristics and flowing can be modeled and simulated using the NSE. The Direct Numerical Simulation (DNS) of NSE with sufficient resolution of the grid size can even resolve turbulence. For the simulation of the fluid flow, we solve two equations numerically, namely the continuity and momentum equation. The Continuity equation

$$\nabla \cdot \vec{u} = 0 \quad \text{where} \quad \vec{u} \quad \text{describes the velocity} \tag{2.1}$$

is derived processesfrom the mass conservation of the flow. For incompressible flows density (ρ) is constant, so the equation reduces to simple divergence free velocity formulation. The Momentum equation

$$\partial_t \vec{u} + \vec{u} \cdot \nabla \vec{u} = -\frac{1}{\rho} \nabla p + \nu \Delta \vec{u} + \vec{F}, \tag{2.2}$$

is derived from Newton's second law of motion applied to fluids. Here $\vec{u}$, p, $\vec{F}$ and ν describes velocity, pressure, force and kinematic viscosity of the fluid respectively. It combines the physics of temporal change, diffusion, convection, pressure forces and source and sink terms. Depending on the properties of the fluid involved and the geometry of the problem, different parts of the equation are more dominant than the other. For instance, for turbulent flows, the inertial/convective part is more dominant than the viscous/diffusive part. Usually, the non-dimensional formulation of the NSE 2.2 is used for all practical purposes, so that we can benchmark with experimental studies. The non-dimensional tensorial formulation of NSE is parametrized by the Reynolds number *Re* and reads,

$$\frac{\partial u_i}{\partial t} + \frac{\partial u_i u_j}{\partial x_j} = -\frac{\partial p}{\partial x_i} + \frac{1}{\mathrm{Re}} \frac{\partial^2 u_i}{\partial x_j^2} + F_i \qquad i = 1, 2, 3 \tag{2.3}$$

The Reynolds number describes the relation of inertial effects to viscous effects. So, it is a very important parameter which characterizes turbulence. We always refer to the above dimensionless form of the NSE for future references.

2.2 Finite difference method

The finite difference method (FDM) is the oldest and simplest and yet powerful method among the discretization techniques for partial differential equations [8]. The derivation and implementation of FDM are particularly simple on structured meshes which are equivalent to a regular Cartesian grid. The nodal value of the approximate solution at node i and time step n

$$\vec{u}_{num}(\vec{x}_i, t^n) \approx \vec{u}(\vec{x}, t) \tag{2.4}$$

is a pointwise approximation to the true solution of the partial differential equation. Taylor series expansions or polynomial fitting techniques are used to approximate all space derivatives in terms of $\vec{u}_{num}(\vec{x}_i, t^n)$ and solution values at a number of neighbouring nodes. For example, if we consider the uniform 1-D mesh given by, the computational

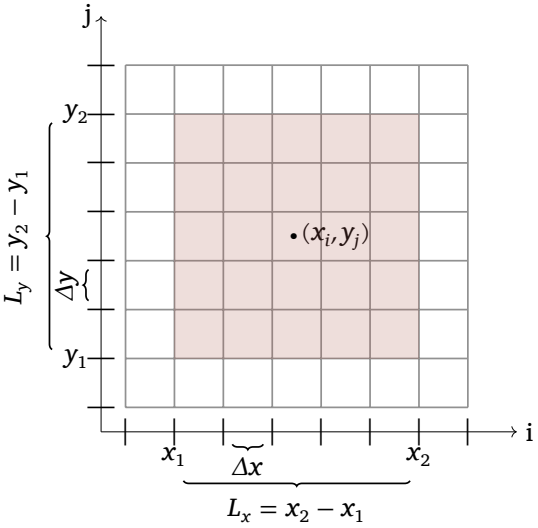


Figure 2.1: The discretization grid.

domain $\Omega = (a, b)$ is an interval. A subdivsion of this interval into N subintervals $\Omega_k = (x_{k-1}, x_k)$ of equal size yields the simplest representation of structured meshes. The $N + 1$ grid points are numbered from left to right. In multi-dimensions,a structured mesh is a net of grid lines (Figure 2.1) with

$$x_i = i\Delta x, \quad \forall\ i = 0, 1, ..., N. \tag{2.5}$$

The first order derivative in 1-D can be approximated by the central difference scheme which is of second order accuracy.

$$\left(\frac{\partial u}{\partial x}\right)_i = \frac{u_{i+1} - u_{i-1}}{2\Delta x}. \tag{2.6}$$

The same principle can be extended for second order derivatives in 1-D as second-order approximation at node *i*.

$$\left(\frac{\partial^2 u}{\partial x^2}\right)_i = \frac{u_{i+1} - 2u_i + u_{i-1}}{(\Delta x)^2}. \tag{2.7}$$

On a nonuniform mesh in 2D, the coefficients are different and must be derived individually for each grid point. Since, the equation 2.3 is time-dependent, suitable choice of time-differencing is important. Explicit, Implicit and Trapezoidal methods are the simplest schemes. More accurate schemes like Range-Kutta could also be employed.

3 Turbulence theory

Turbulence can be described as fluid moving in complex and indeterministic way. Its very difficult to predict the future state of turbulent flows due to its complexity. Turbulence

Figure 3.1: Turbulence in coffee mixing (left) and Jupiters turbulent atmosphere (right).

is ubiquitous, its present everywhere from turbulent flow of blood in our arteries to flow of rapid rivers to atmospheric dispersion of pollutants to vortices in the atmosphere of Jupiter (Figures 3.1a and 3.1b).

For the past 150 years, the greatest minds in science put their time and efforts to quantify this chaotic nature of turbulence. Werner Heisenberg, German theoretical physicist who worked on the mathematical equations for transition from laminar to turbulent flow once said, “When I meet God, I am going to ask him two questions: Why relativity? And why turbulence? I really believe he will have an answer for the first” [9]. Even Richard Feynman, the famous American theoretical physicist know for his works on quantum mechanics and superfluidity acknowledged turbulence as the most important unsolved problem of classical physics [5].

In a sense, the problem of turbulence was solved long ago. The NSE equations describing the fluid motion were formulated 150 years ago. The Direct Numerical simulation (DNS) of the NSE can resolve turbulence, but for problems concerning real world applications, the solution would take more than a year for its solution. So we would like to resort to have a more efficient way to deal with turbulence. In this direction, Reynolds Average Navier-Stokes (RANS) and Large Eddy Simulations (LES) are the two most common ways to tackle the problem of turbulence with reasonable compute time. In the next section, we present the idea and theory behind LES based on Subgrid-Scale (SGS) modeling.

3.1 Large Eddy Simulation based on Subgrid-Scale modeling

Before we can go into the theory of turbulence and LES, it is prudent to contemplate on the question “Why not use Direct Numerical Simulation of NSE?”. DNS of NSE 2.2 involves no modelling approximation, so a fine grid approximation of NSE by DNS would adequately capture the most accurate physics including turbulence [4]. But when it comes to solving problems of real world applications DNS simulations can take months because of the fine step size required both in time and space. So to have a good trade-off

between accuracy and efficiency, we resort to more modeling approximation which can leverage the information from the large scales to quantify the scale dynamics. This is the basis for Large Eddy Simulations based on Subgrid-Scale modeling.

SGS modeling is based on decomposition of flow into low-frequency resolved modes and high-frequency unresolved modes (Subgrid-Scale stresses). This scale separation is achieved by applying a filter, in our case the finite difference grid itself acts as an implicit filter. The implicit filtering can be achieved by using discrete filters based on the stencil/grid provided by the finite difference method. The discrete filters can be developed in 1-D and can be extended to 3-D by construction of product and derivative rules.

In Large Eddy Simulation, the flow variables are divided into a low frequency resolvable part and a high frequency modeled part by applying a convolution kernel:

$$\tilde{f}(x) = \int G(x,x')f(x')\,\mathrm{d}x' \quad \text{resulting in} \quad u_i = \tilde{u}_i + u_i', \tag{3.1}$$

where $\tilde{u}_i$ is the resolvable large scale motion and u_i' is the modeled Subgrid-Scale part.

The filter function $G(x,x')$ depends on the filter width Δ and satisfying the normalization condition

$$\int G(x,x')dx' = 1. \tag{3.2}$$

Some common isotropic filters are the box filter, defined [12] as

$$G(x,x') = \begin{cases} \frac{1}{\widetilde{\Delta}}, & \text{if} \quad |x-x'| \leq \frac{\widetilde{\Delta}}{2} \\ 0, & \text{otherwise} \end{cases} \tag{3.3}$$

and the Gaussian filter

$$G(x,x') = \left(\frac{6}{\pi\widetilde{\Delta}^2}\right)^{\frac{1}{2}} \exp\left(-\frac{6|r|^2}{\widetilde{\Delta}^2}\right). \tag{3.4}$$

Applying any one of the filtering operator to the NSE 2.3, we get the filtered form of NSE.

$$\frac{\partial\widetilde{u_i}}{\partial t} + \frac{\partial\widetilde{u_i}\widetilde{u_j}}{\partial x_j} = -\frac{\partial\tilde{p}}{\partial x_i} + \frac{1}{\mathrm{Re}}\frac{\partial^2\widetilde{u_i}}{\partial x_j^2} - \frac{\partial\tau_{ij}}{\partial x_j} + F_i \tag{3.5}$$

where $\tau_{ij} = \widetilde{u_iu_j} - \widetilde{u_i}\widetilde{u_j}$ is the LES Subgrid-Scale stress.

An additional term appears on the RHS due to the non-linear convection term. This term incorporates the information of the small scales into the filtered NSE. This is schematically shown in Figure 3.2 where the energy $E(k)$ is plotted against the wavenumber k.

Modeling τ_{ij} to include the small scale dynamics is the objective of SGS. In 1887, Boussinesq [3] postulated that the momentum transfer caused by the turbulent eddies

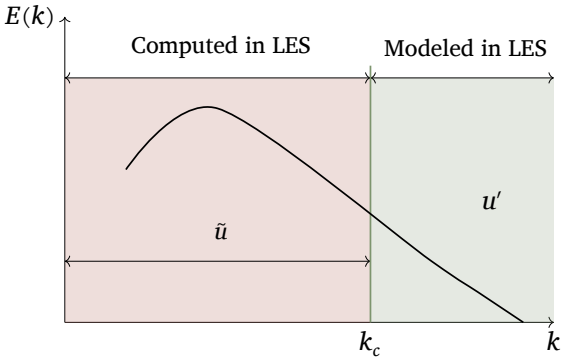


Figure 3.2: Energy spectrum showing cut-off wavenumber k_c separating resolved and modeled scales [11].

can be modeled with eddy viscosity. The mathematical structure is similar to that of molecular diffusion described by molecular viscosity.

$$\tau_{ij} - \frac{1}{3}\tau_{kk}\delta_{ij} = -2\nu_t\tilde{S} \tag{3.6}$$

where $\tilde{S}$ is the large scale strain-rate tensor and ν_t is the SGS turbulent viscosity/eddy viscosity. This eddy viscosity ν_t is fed to Navier-Stokes through the effective viscosity $\nu_{\text{eff}} = \nu_{\text{mol}} + \nu_t$.

3.2 Constant Smagorinsky

The first SGS model developed to model the eddy viscosity ν_t is the Smagorinsky-Lilly model [13]. It models the eddy viscosity as:

$$\nu_t = (C_s\Delta)^2 \left|\tilde{S}\right|, \tag{3.7}$$

where $\Delta = V^{\frac{1}{3}}$ with $V = (\Delta x \Delta y \Delta z)$ and

$$\left|\tilde{S}\right| = \sqrt{2\widetilde{S_{ij}}\widetilde{S_{ij}}} \quad \text{with} \quad \widetilde{S_{ij}} = \frac{1}{2}\left(\frac{\partial\widetilde{u_i}}{\partial x_j} + \frac{\partial\widetilde{u_j}}{\partial x_i}\right) \tag{3.8}$$

The most important challenge for the Smagorinsky models is determining the parameters C_s with given Δ. It is the parameter controlling the rate of kinetic energy dissipation happening at small scales. C_s is given explicitly in the formulation and a value between $0.1 - 0.2$ has been found to yield good results for wide range of flows except for wall bounded flows. Regardless, this simple scheme suffers from high dissipative characteristics and is not so robust parameters with a non-adaptive value of C_s. This may lead to some undesirable and non-physical behaviour of the system.

3.3 Dynamic Smagorinsky

Germano and Lilly proposed in [7] a procedure where the Smagorinsky model constant C_s is computed dynamically based on the information of the resolved scales $\tilde{u}$. The idea

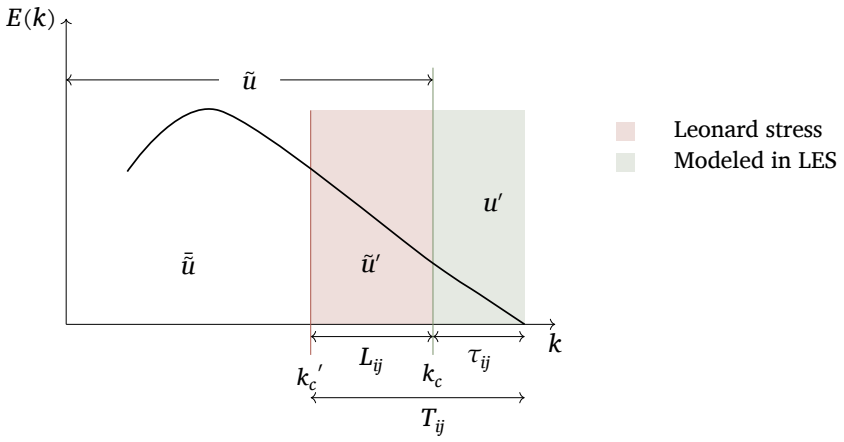


Figure 3.3: Energy spectrum showing cut-off wavenumbers for filtered k_c and $k_c{}'$ test filtered quantities [11].

is to apply a filter for the second time to the equations of motion, i.e. momentum and continuity equations. The new explicit filter width $\bar{\Delta}$ is greater than that of $\tilde{\Delta}$, usually twice that of the grid filter. Both filters produce a resolved field. The difference between the contribution of these fields is the resolved turbulent stress usually referred as Leonard stress L_{ij} which is used to compute C_s adaptively. In Figure 3.3 the orange coloured region represents the Leonard stresses L_{ij} which are defined by Germano [6] as

$$L_{ij} = T_{ij} - \overline{\tau}_{ij} = \widetilde{\tilde{u}_i \tilde{u}_j} - \bar{\tilde{u}}_i \bar{\tilde{u}}_j \tag{3.9}$$

where $T_{ij} = \widetilde{\tilde{u}_i \tilde{u}_j} - \bar{\tilde{u}}_i \bar{\tilde{u}}_j$ is the residual stress tensor of the test filter and $\overline{\tau}_{ij} = \widetilde{\tilde{u}_i \tilde{u}_j} - \widetilde{\tilde{u}_i}\widetilde{\tilde{u}_j}$ is the Subgrid-Scale stress of grid filter that is test filtered again.

By applying Smagorinsky model for stresses in the Germano identity 3.9 and minimizing the error associated due to modeling in the least-square sense, we get an expression for the dynamic C_s, i.e.

$$C_s^2 = \frac{L_{ij} M_{ij}}{M_{ij} M_{ij}}, \tag{3.10}$$

where $M_{ij} = 2\tilde{\Delta}^2 \left(\widetilde{|\tilde{S}| \tilde{S}_{ij}} - \alpha^2 |\bar{\tilde{S}}| \widetilde{\bar{S}_{ij}} \right)$ and $\alpha = \bar{\Delta}/\tilde{\Delta}$.

However, expression 3.10 can be numerically unstable since the numerator could become negative and large fluctuations in C_s are observed. Hence, additional averaging of the error is done, resulting in

$$C_s^2 = \frac{\langle L_{ij} M_{ij} \rangle}{\langle M_{ij} M_{ij} \rangle}. \tag{3.11}$$

The averaging can be as simple as volume averaging with respect to the neighbours or spatial averaging over the entire domain. For details about the explicit filters used during the simulations are described in the following paragraphs.

3.3.1 Finite difference method as Implicit filter

From the wavenumber-dependent characteristics of the errors in finite difference scheme, the hidden filtering properties of the scheme can be seen nicely by comparing the equivalence between a finite difference and the exact derivative of the filtered variable :

$$\left(\frac{\partial u}{\partial x}\right)_i \approx \frac{u_{i+1} - u_{i-1}}{2\Delta x} = \frac{\partial}{\partial x}\int_{x_{i-1}}^{x_{i+1}} u\,\mathrm{d}x = \frac{\partial \bar{u}}{\partial x}. \tag{3.12}$$

3.3.2 1-D test filters

The test filter width is usually twice the size of the grid filter i.e. $\bar{\Delta} = 2\tilde{\Delta}$.

$$\bar{f}(x) = \frac{1}{2\tilde{\Delta}}\int_{-\tilde{\Delta}}^{+\tilde{\Delta}} \tilde{f}(x')dx' \quad \text{where} \quad \bar{\Delta} = 2\tilde{\Delta} \tag{3.13}$$

where $\widetilde{\Delta}$ is the grid filter width. Numerical integration of the above equation leads to various forms of discrete filters [10]. The integration can be done by trapezoidal 3.14 and Simpsons rule 3.15. Equation 3.14 is used most extensively in LES based on finite difference approximations [2, 10].

Trapezoidal rule :

$$\bar{f}(x) = \frac{1}{4}(\tilde{f}_{j-1} + 2\tilde{f}_j + \tilde{f}_{j+1}) \tag{3.14}$$

Simpson's rule :

$$\bar{f}(x) = \frac{1}{6}(\tilde{f}_{j-1} + 4\tilde{f}_j + \tilde{f}_{j+1}) \tag{3.15}$$

Discrete filters of higher order can also be constructed from using stencils beyond $(j-1)$ and $(j+1)$.

Seven point filter :

$$\bar{f}_j = \frac{1}{256}\left(\tilde{f}_{j-3} - 18\tilde{f}_{j-2} + 63\tilde{f}_{j-1} + 164\tilde{f}_j + 63\tilde{f}_{j+1} - 18\tilde{f}_{j+2} + \tilde{f}_{j+3}\right) \tag{3.16}$$

3.3.3 Extension to 3D – Construction by product

Multi-dimensional filters can be defined by the composition of one-dimensional filters applied in each space direction.

$$F^p = \prod_{i=1}^{p} F^i, \tag{3.17}$$

where p is the dimension. This product is equivalent to a sequential application of the one-dimensional filter. The resulting filter is a $(2N+1)^3$ -point stencil. In the three-dimensional case, the discrete operator reads

$$\bar{f}_{i,j,k} = F^p f_{i,j,k} = \sum_{l=-N}^{N}\sum_{m=-N}^{N}\sum_{n=-N}^{N} a_l a_m a_n f_{i+l,j+m,k+n}, \tag{3.18}$$

where a_l, $a_m$and a_n are the coefficients of the discrete filter.

The Dynamic Smagorinsky model is self contained i.e there is no need to specify the Smagorinsky constant C_s. But when it comes to the matter of stability the model can produce parameters which are variant 10 times from the mean and the denominator can potential go to zero resulting in instability. It also produces negative turbulent viscosities which the advocates of the model believe can capture the effects of backscatter: energy transfer from small scales to large scales.

4 Acceleration using OpenACC

OpenACC is a directive based programming paradigm for accelerating programs with simple syntax and minimal change to the program. It is designed for performance portability across devices and platforms. Simple insertions of directives into the code will inform the OpenACC compiler about possible acceleration by GPU or multicore.

The three most common used constructs in OpenACC are:

1. **Data construct**: An accelerator data construct defines a region of the program within which data is accessible by the accelerator.

2. **Parallel construct**: The parallel construct launches a number of gangs (thread-blocks) executing in parallel.

3. **Kernel construct**: The OpenACC compiler essentially translates the loops into a kernel that can run in parallel on the accelerator.

Both parallel and kernel constructs try to exploit the loop parallelism and map it to device parallelism. The most important difference between them is that, kernel operation is more implicit, giving the compiler more freedom to identify and map parallelism. The parallel construct is more explicit, and requires more explicit syntax from the programmer to determine when it is appropriate.

5 Results

5.1 Finite difference filter for high frequencies

Explicit filtering by finite difference with different filter width where conducted to demonstrate the smoothing properties of the scheme.

A seven point discrete filter operator in 1-D can be constructed as following.

$$\overline{f}_j = \frac{1}{256}\left(\tilde{f}_{j-3} - 18\tilde{f}_{j-2} + 63\tilde{f}_{j-1} + 164\tilde{f}_j + 63\tilde{f}_{j+1} - 18\tilde{f}_{j+2} + \tilde{f}_{j+3}\right) \tag{5.1}$$

Extension to multi-dimensions can be achieved by the construction of a product as shown in equation 3.18.

$$\overline{f}_{i,j,k} = F^p\tilde{f}_{i,j,k} = \sum_{l=-3}^{3}\sum_{m=-3}^{3}\sum_{n=-3}^{3} a_l a_m a_n \tilde{f}_{i+l,j+m,k+n} \tag{5.2}$$

where $a_1 = \frac{1}{256}, a_2 = -\frac{18}{256}, a_3 = \frac{63}{256}, a_4 = \frac{164}{256}, a_5 = \frac{63}{256}, a_6 = -\frac{18}{256}, a_7 = \frac{1}{256}$.

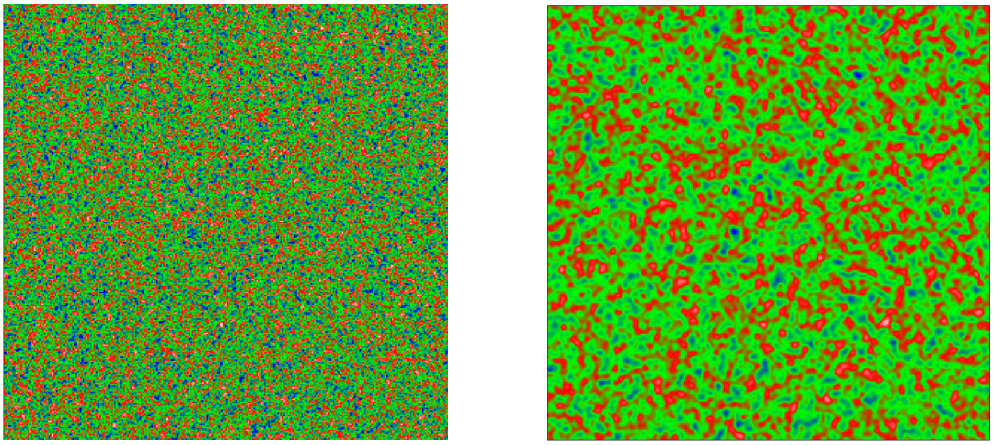


Figure 5.1: Random noise (left) and filtered with seven point filter.

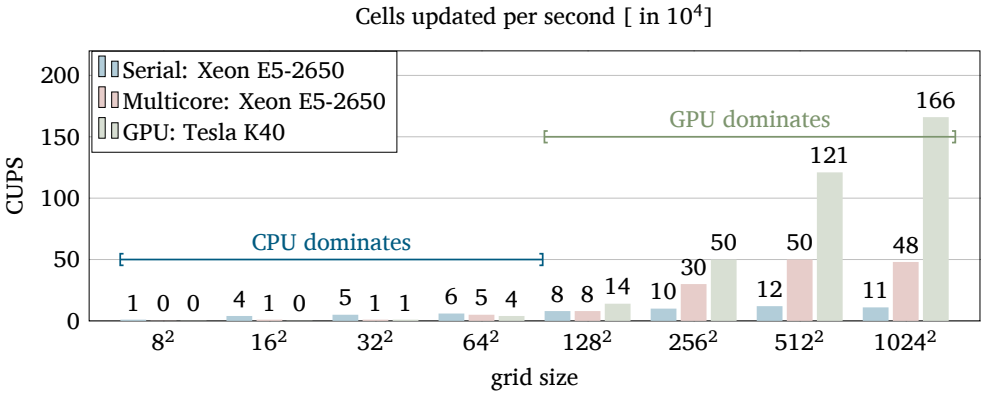


Figure 5.2: Speedup of 4× for 1024^2 cells using Implicit Jacobi.

5.2 Qualitative studies to show the speedup

5.2.1 Diffusion

The diffusion equation is solved in a square geometry $\Omega = [0, 2]^2$ with Dirichlet boundary conditions i.e. $u = 0$ at $\partial\Omega$ and initial condition given by $u_0 = u(t = 0)$. The Implicit Jacobi method is used to solve the discretized equation. The solution domain is initialized *ExpSinus* and the solution is compared with the analytical solution. At a system size of 128×128, the GPU acceleration performs better than serial and multi-core implementation. At system size of 1024×1024, the GPU implementation is 4 times faster than the multicore implementation.

5.2.2 Constant Smagorinsky

The diffusion equation is again solved in a square geometry with the same Dirichlet boundary conditions and initial conditions but with the viscosity being the effective viscosity calculated by $\nu_{\text{eff}} = \nu_{\text{mol}} + \nu_t$. A C_s value of 0.1 is used in all the simulations.

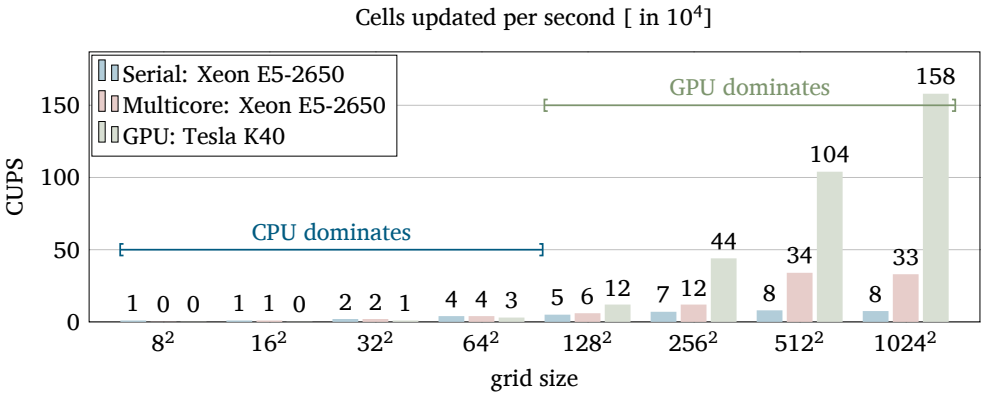


Figure 5.3: Speedup of 5× for 1024^2 cells using Constant Smagorinsky.

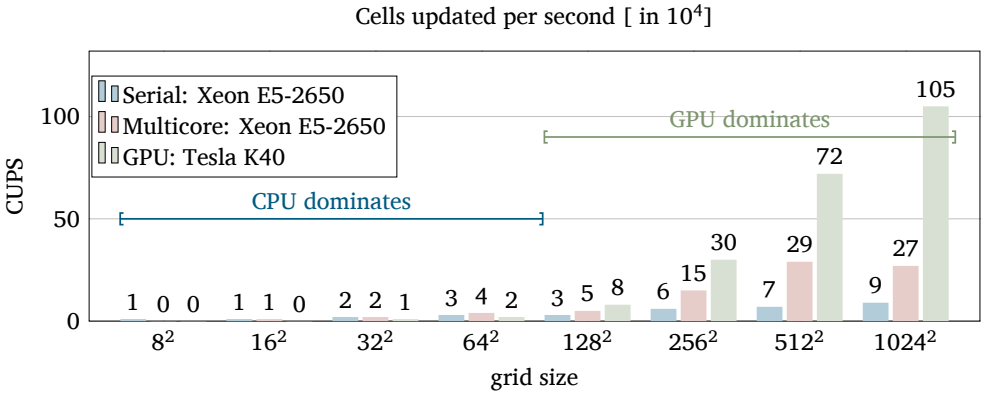


Figure 5.4: Speedup of 4× for 1024^2 cells using Dynamic Smagorinsky.

Essentially the simulation runs at higher viscosity with the constant smagorinsky model feeding the local varying effective viscosity to the fluid-solver. At a system size of 128×128 , the GPU acceleration performs better than serial and multi-core implementation. At system size of 1024×1024, the GPU implementation is 4 times faster than the multi-core implementation.

5.2.3 Dynamic Smagorinsky

With the same set-up as that of Constant Smagorinsky, the dynamic Smagorinsky calculates the C_s dynamically. The negative eddy viscosities are clipped to zero. Thus the simulation runs at higher viscosity with the Constant Smagorinsky model feeding the local varying effective viscosity to the fluid-solver. At a system size of 128×128, the OpenACC acceleration performs better than serial and multi-core implementation. At system size of 1024×1024, the GPU implementation is 4 times faster than the multi-core implementation.

6 Conclusion and Outlook

Two most common turbulence models namely Constant Smagorinsky and Dynamic Smagorinsky based on SubGrid-Scale modelling are implemented. The eddy viscosity computed locally in the turbulence modules was used to calculate the effective viscosity. The fluid-solver is then used with this locally varying effective viscosity with Dirichlet boundary conditions to solve the NSE. In the case of the Dynamic Smagorinsky model, explicit filtering is included by using discrete filters based on top-hat filters. Some negative viscosities in Dynamic Smagorinsky model are observed but are clipped to zero for stability reasons. Due to the inherent local stencil operation of finite differencing, the code was effectively ported to GPU through OpenACC. Qualitative benchmark studies for parameter boundedness and the convergence to zero for decreasing grid width were conducted. Performance comparisons for serial, multicore and OpenACC were realized.

The future works should focus more on the quantitative benchmark studies for different boundary conditions. Tuning of the C_s is needed for stability reasons in Constant Smagorinsky. Since the Smagorinsky models are highly dissipative in nature, more matured models like mixed models based on linear combination of Smagorinsky and scale similarities models should be developed.

7 Acknowledgements

I would first like to thank Ivo Kabadshow for giving me the opportunity to participate in this exciting workshop. My heart-full thanks to my supervisor Anne Severt for giving me the freedom and independence in working on this exciting project. I wish her the very best in her scientific pursuits. I would like to thank my fellow colleagues for making these 2 months a very memorable time of my life.

References

[1] G. K. Batchelor. *An introduction to fluid dynamics*. Cambridge university press, 2000.

[2] E. Balaras, C. Benocci, and U. Piomelli. *Finite-difference computations of high Reynolds number flows using the dynamic subgrid-scale model*, volume 7. Springer, 1995.

[3] J. Boussinesq. *Essai sur la théorie des eaux courantes. Mémoire des Savants étrangers*. 1877.

[4] A. Dewan. *Tackling turbulent flows in engineering*. Springer Science & Business Media, 2010.

[5] I. Eames and J. Flor. *New developments in understanding interfacial processes in turbulent flows*, volume 369. The Royal Society, 2011.

[6] M. Germano. *Turbulence: the filtering approach*, volume 238. Cambridge Univ Press, 1992.

[7] M. Germano, U. Piomelli, P. Moin, and W. H. Cabot. *A dynamic subgrid-scale eddy viscosity model*, volume 3. AIP Publishing, 1991.

[8] C. Grossmann, H.-G. Roos, and M. Stynes. *Numerical treatment of partial differential equations*. Springer, 2007.

[9] A. Marshak and A. Davis. *3D radiative transfer in cloudy atmospheres*. Springer Science & Business Media, 2005.

[10] F. Najjar and D. Tafti. *Study of discrete test filters and finite difference approximations for the dynamic subgrid-scale stress model*, volume 8. AIP Publishing, 1996.

[11] P. Sagaut. *Large eddy simulation for incompressible flows: an introduction*. Springer Science & Business Media, 2006.

[12] P. Sagaut and R. Grohens. *Discrete filters for large eddy simulation*, volume 31. Wiley Online Library, 1999.

[13] J. Smagorinsky. *General circulation experiments with the primitive equations: The basic experiment*, volume 91. 1963.

[14] G. H. Yeoh and K. K. Yuen. *Computational fluid dynamics in fire engineering: theory, modelling and practice*. Butterworth-Heinemann, 2009.

Sorting and Administration of Particles in OpenCL

Utkan Çalişkan
Computational Science and Engineering
Istanbul Technical University
Turkey
caliskanut@itu.edu.tr

Abstract *Two parallel implementations of neighbor list techniques for particle-based simulations are presented. The first technique is based on a linked-cell approach, that is commonly used in Molecular Dynamics, whereas the second is a container-based approach, which collects particles in chunks of finite sizes representing cells. Both algorithms were implemented in OpenCL in order to achieve portability between different compute architectures. For this project the implementations were compared on Intel Xeon Phi and NVIDIA GPU.*

1 Motivation

Simulation methods based on the application of particles are complementary techniques to mesh-based simulation approaches often used to solve partial differential equations (PDEs) or to simulate the behavior of complex statistical systems. Especially for cases where the system under study is atomistically resolved or composed of coarse-grain agglomerates, particles are often a natural choice to map the physical system onto a numerical method. In particular atomistic systems composed of N particles are often simulated by Molecular Dynamics (MD) or Monte Carlo (MC) techniques in order to study their physical or chemical behavior. Most often it is of crucial imporantance to compute the interaction between particles, which can often be reduced to pair-wise interactions between particles i and j. In such cases the total force F_i on particle i can be computed as the superposition of individual forces between particles pairs f_{ij}:

$$F_i = \sum_{i \neq j} f_{ij} \tag{1.1}$$

Since interactions have to be evaluated between all particle pairs, the computational complexity is $O(N^2)$. Although the complexity prefactor can be reduced by a factor of two when applying Newton's third law, the computational complexity remains unchanged.

$$f_{ij} = -f_{ji} \tag{1.2}$$

Due to the mobility of the paricles, this complexity is even true for short range interactions, since the environment of each particle is not static and the neighbor relations between particle pairs have to be updated (re-computed) in every time step. These short-range interactions, which are considered in this report, are characterized by a rapidly decreasing interaction potential between particles which usually can be set to zero outside of a spherical range of influence with radius R_c called the cut-off radius.

In order to effectively reduce the complexity from $O(N^2)$ to $O(N)$ techniques have to be developed, that keep track of neighborhood relations between particles, i.e. linear scaling neighbor list techniques. Two possible implementations of these neighbor list techniques are described in this report.

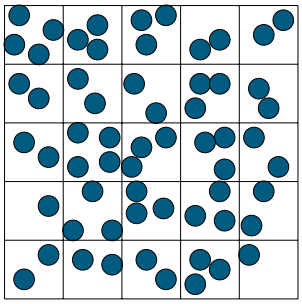


Figure 2.1: System of particles subdivided into cells.

2 Sorting Techniques

As discussed before, cut-off radii can be used in MD sumulations based on short-range interactions in order to reduce the number of computed pair-wise particle-particle interactions and pair-wise distances checks. To avoid testing mutual pair distances between all particle pairs in the system, sorting or grouping techniques are required, that provide information for each particle i about all particles j located in close spatial proximity.[8]

The most simple way to provide this information is to sort particles into cubic cells of side length $L_c = L/N_c$, where L is the length of the system and N_c is the number of cells in a cartesian direction defined as $N_c = L/R_c$ (integer division). Using this type of organization in cells, it can be assured that for a given particle located in cell (i_x, i_y, i_z) all required interaction partners are situated in surrounding cell $(i_x \pm 1, i_y \pm 1, i_z \pm 1)$. Therefore instead of needing to check the distances to all other particles in the system, only the distances to the particles in the surrounding eight (2D case) or 26 (3D case) cells need to be computed and evaluated. Figure 2.1 represents such a subdivision.

For the creation of the particle subdivision into cells, two different approaches will be described hereafter: (i) a linked-cell approach, where the particles are stored in an unsorted array and the cells are represented by linked-lists and (ii) a container-based approach, where particles representing a cell are stored in contiguous memory locations. In both cases the cell-length is assumed to be the length of the cut-off radius for a fictitious short-range potential, as that would allow to restrict the number of particle-particle distance calculations to all particles within bordering cells. Applying this procedure, the computational complexity is reduced from $O(N^2)$ to $O(NM)$ where the prefactor $M \ll N$ is dependent on the density.

2.1 Linked-Cell List

In order to organize the particles into a linked-cell list, two arrays of integers are required. The first array is of size N and contains the linked-lists describing the cells. In a second array, the size of which is the number of cells in the system, the entry particles for each cell list are stored. Particles are stored in a seperate array without any further sorting applied to them. To group the particles into cells, for each particle the cell it is located in is calculated and the particle index in the particle array is appended to the

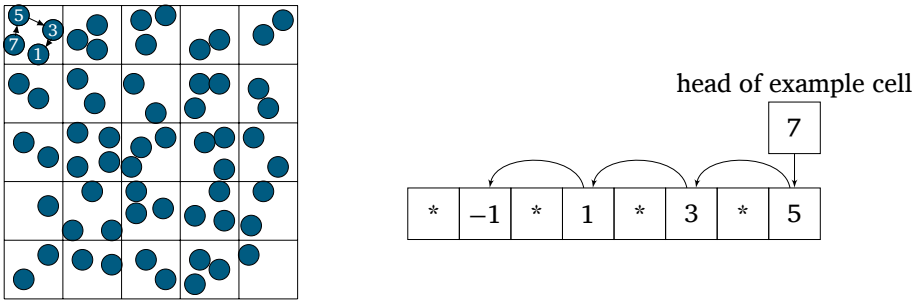


Figure 2.2: llustration of linked-cell approach.

linked list describing that cell. This is achieved by setting the list entry with the same index as the particle to the current entry value of the cell and then updating the cell entry to be index of the current particle. This results in a computational effort of $O(N)$ for the grouping into cells. As the entry values form the head of the linked lists, the corresponding array is called 'head' in this report. Figure 2.2 shows the result of such a grouping procedure, in 2.2a the spatial division into cells can be seen, whereas in figure 2.2b the representation of the first cell in memory is shown. The content of the head array points to the index of the first particle within the cell. To find all particles in the cell it is required to follow the sequence of indices stored in the list array until an invalid index (in this case -1) is encountered, marking the end of the linked list. To access the particles, the stored indices are used as indices of the seperate particles array.

Interactions are taken into account between particles which are within the cut-off sphere. Since the cell size is chosen according to R_c i.e. $R_c \leq L_c$, the particles inside neighboring cells are checked for force calculations. The memory requirement for the arrays is each linearly dependent on the number of particles and cells in the system. The entries have to be updated in every MD simulation time-step since there will be a particle fluctuation across the cell boundaries.

In terms of parallelization of the algorithm with OpenCL, threads are assigned in 1D range which means threads will share particles from 0 to N either in a blockwise fashion (contiguous index range of particles) or in a cyclicwise fashion (jumping between particle indices with a stride of n_{th} where n_{th} is the number of threads. While creating the linked-cell list in parallel, race conditions may occur because two or more different threads can try to modify the same cell at the same time by trying to change the value of the 'head' array for the given cell. This may lead to lost particles, as overwritten values of 'head' may not be processed in respective concurrent updates. In order to avoid race conditions Compare and Swap (CAS) atomic operations, locks or copies can be used. For further information on locks and copies in order to parallelize the creation of linked-cell lists via OpenMP, see Jongmanns[5].

2.2 Container-Based Sorting

In contrast to the linked-list based approach, where the cells are represented by linked-lists and the particles remain in a seperate array, in the container-based variant, the particles are physically connected to a cell as each cell is represented by a contigous

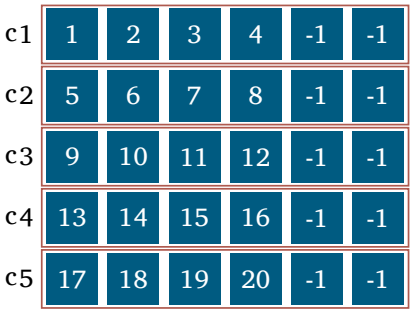


Figure 2.3: Illustration of the initial step container-based sorting, items inside the red rectangle represent particles located in the given cell.

memory space, in which particles contained in the cell are stored. For the sake of simplicity particles are created inside their respective cell in this project, to avoid the additional need for a first-step sorting algorithm. In figure 2.3 the 2D-array storing the particles is shown. Each red box, i.e. each line, represents a single cell with its content. As the figure shows the system after initialization, each cell contains the same number of particles in ascending order. As particle fluctuation is expected, the memory space reserved for each cell is larger than the number of particles in the first steps requires. Unused particle positions are marked with invalid indices, usually negative values like '−1'.

In order to update the cell contents, a two-step approach is chosen that can be reasonably well implemented in OpenCL. In the first step particles that left the local cell are stored in a cell-owned buffer and are removed from the local cell by marking them with an invalid index. Afterwards in a second step for each cell the buffers of the surrounding cells are checked for particles that moved into the given cell. If such a particle is found, it is inserted into an empty location, i.e. a location containing a particle with invalid index. In an OpenCL implementation a 3D range of work-items can be used to work on the cells. Each work-item can indepentently fill the cell-based buffer in the first step and in the second step each work-item can check the buffers of the surrounding cells without interference from other work-items, as write operations only take place in the memory assigned to the local work-item. Another variant, that was not investigated in depth within the project, writes particles directly into remote cells, once they leave the local cell. This would require synchronization with atomic operations, as inter work-group synchronization would be required.

3 Implementation

OpenCL (Open Computing Language) is an open, royalty-free standard for cross-platform, parallel programming of diverse processors [6].

We used OpenCL for a parallel implementation of many-particle sorting algorithms in order to run this code on CPUs, GPUs, and MICs. OpenCL comprises a C-API for platform and runtime management on the host and a special programming language for the device code. Devices contain a large number of processing elements or execution units for integer and floating- point operations. These units are organized within so-called

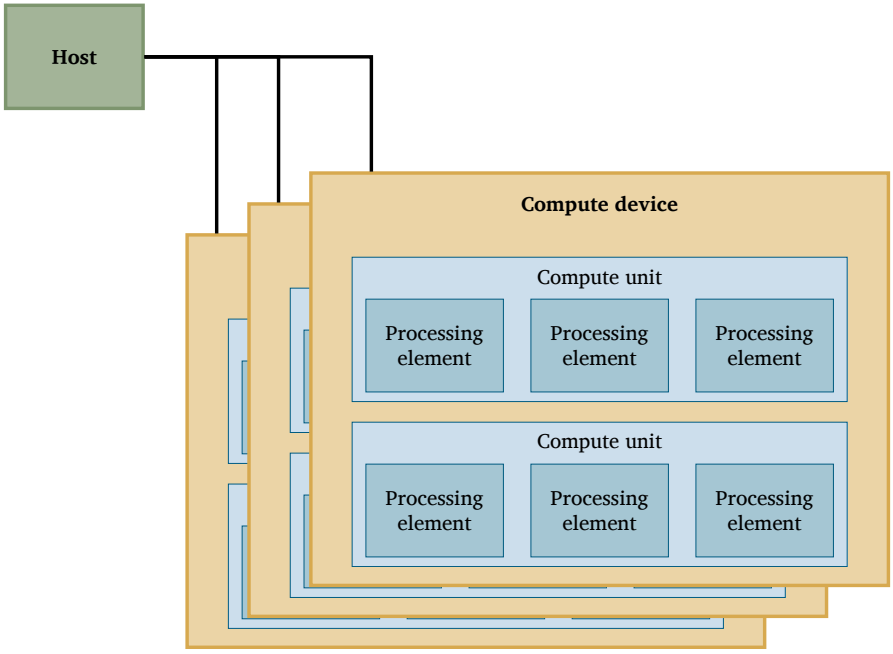


Figure 3.1: OpenCL Platform Model.

compute units or multi-processors and SIMD-cores. In OpenCL nomenclature, a single unit of execution is called a “work-item” or “thread”. A “work-group” contains a larger number of work-items which are scheduled together for execution on the same multi-processor. Brief structure of an OpenCL application with external device is illustrated in Figure 3.1 The memory architecture is hierarchical: The most important memory types are local and global memory. Each multi-processor has its own local memory store which is very fast and can be used as a programmer-managed cache. However, it is strongly limited in size. In contrast, global memory is large but slow. Usually, the main memory of the GPU (both local and global) is denoted as “device memory” and the main memory of the CPU as “host memory”. The large number of threads which run in parallel on a device puts strong pressure on the memory bandwidth. For this reason, it is an important programming challenge to carefully design memory access patterns so that accesses to global memory can be coalesced.

Both MD algorithms are implemented in OpenCL. As it mentioned before, linked-cell list algorithm has one-dimensional thread ranging while container-based algorithm has three-dimensinal thread ranging.

3.1 Linked Cell List

The implementation of the linked-cell list alogrithm emulates a MD simulation, consisting of particle creation, linked-cell list creation, force calculation and particle movement. These steps are handled in individual kernels, as seen in algorithm 1. For initialization the particles are created with a random distribution within the system. Afterwards the

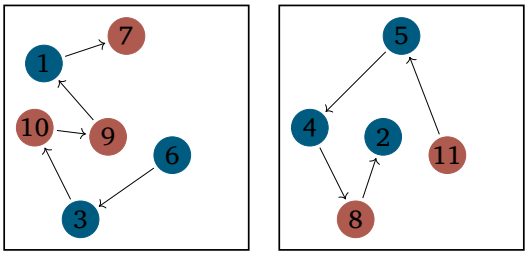


Figure 3.2: Two sets of particles (1-6), (7-11) administrated by two different threads (red, blue) and sorted into two cells. Due to the concurrency of the threads, it is not guaranteed that the cell list contain monotonously descending particle indices, as particles are added to the list from both threads at the same time. With regard to each thread, the lists are monotonously descreasing, like in the serial case.

particles are grouped into the linked-cells, that are the basis for the force calculation in the next step. Once the forces are calculated, the particle are moved to new positions based on the results of the force calculation. This procedure is repeated N_{step} times.

Algorithm 1 Main program for MD with Linked-Cell List.

```
1: createParticles()
2: for i = 0 → N_Step do
3:     createLcList();
4:     forceCalculation();
5:     moveParticles();
6: end for
```

The parallel implementation of the creation of linked-cell lists is presented in [3]. Within this project the linked-cell lists were implemented with the OpenCL framework, the kernels use a 1D work-item range, distributing chunks of the particle array onto the kernels. As previously mentioned the parallel implementation of linked-cells leads to race-conditions, as different work-items are bound to sort particles into the same cell at the same time, see 3.2. Since no inter work-group synchronization is availble in OpenCL, atomic operations are required to solve this problem. To ensure the integrety of the created lists, *atomic_cmpxchg*() was used.

Algorithm 2 Linked Cell List creation subroutine.

```
1: repeat
2:     old = lc_ cells[cellidx][HEAD];
3:     lc_ cons[gid] = old;
4:     result = atomic_ cmpxchg(lc_ cells[cellidx][HEAD], old, gid);
5: until (old != result);
```

Algorithm 2 shows how the linked cell creation with compare and swap operations *atomic_cmpxchg*() works: As described in the previous chapter, two arrays are required

for the representation of the cells. `lc_cells[ic][HEAD]` contains the 'head' values, i.e. the entry point for cell ic, and `lc_cons` contains the connections between the particles, i.e. the linked lists. Both arrays are initialized with invalied indices (-1). In order to assign a particle to a cell in OpenCL a cell index cellidx is calculated. Then the old entry value is temporarily stored to old. Afterwards the connection of the local particle is set to this old entry value. In the last step the compare and swap operation is executed which ensures that the entry value is only updated if it was not changed since it was stored to the old value. If the value would have changed, that means that a different work-item has added another particle to the cell in the meanwhile, so the assigning procedure has to be repeated. This ensures correct lists, containing all the particles at the end of the procedure.

3.2 Container-Based Sorting

In contrast to the linked-cell list version, the ideal implementation of the container-based sorting would consider contiguous chunks of memory which are assigned to every cell and which represent the containers to host the particles within cells. In the present implementation, however, a simplified approach is taken. Here we do not assign contiguous space to each cell but sort particles according to their global index in increasing order. Although this allows a simple access of particles in each cell it has as a consequence a random jump in global memory. When implementing the container-based variant in OpenCL it is useful to assign a single cell or clusters of neighbored cells to a single work-item. The range of the work-item then may be three-dimensional in order to have representation that is similar to the cell structure. For the algorithm additional space must be allocated for each cell to account for particle fluctuation, also a buffer for the exchange of particles in parallel must be provided.

Algorithm 3 Main program for MD with container-based algorithm.

```
1: createParticles()
2: for i = 0 → N_Step do
3:     forceCalculation();
4:     movingParticles();
5:     buffering();
6:     assignParticles();
7: end for
```

The particlesInCells array contains the memory chunks representing the cells. Algorithm 3 gives an overview about the needed OpenCL kernels. In the createParticles() kernel the particles are created in the particlesInCells array in an ascending order (see also figure 2.3). In contrast to the linked-cell version, the force calculation can be processed immediatly afterwards, as the particles are already sorted into cells (forceCalculation()). Afterwards the particles are moved according to the results of the force computations (movingParticles()). To finalize the time step, now buffering and reassignment of the particles has to take place. Figure 3.3 shows which particles left their respective cells and need to be copied into buffers before further processing. In figure 3.4 it can be seen, that the outgoing particles are copied to a buffer and their places in the arrays are filled with invalid indices to mark them as free. This finishes the buffering

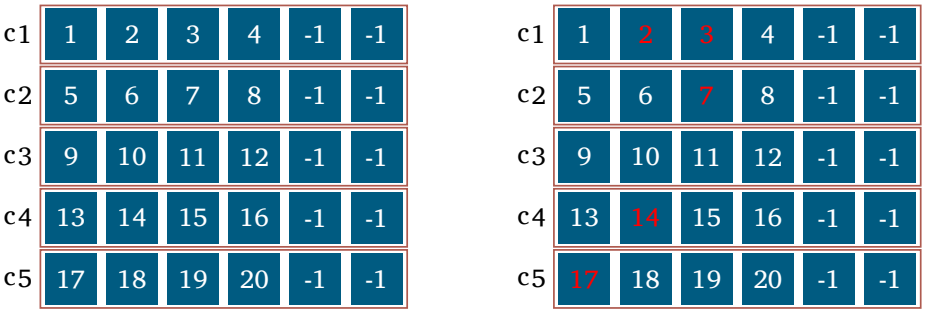


Figure 3.3: The structure of the array – the leaving particles are marked as red.

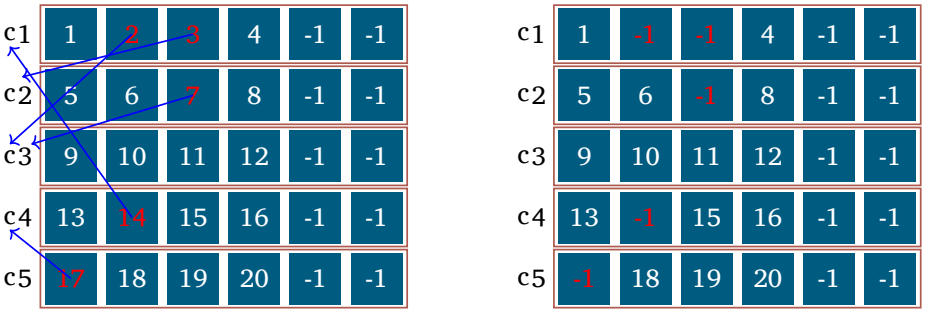


Figure 3.4: Leaving particles are marked as red and the directions of those particles with blue arrows.

step. Now the reassignment of the particles takes place by checking the buffers of cells neighboring the local cell. All particles that are moved into the local cell are inserted into empty spaces of the local array. Figure 3.5 shows a possible final state after the reassignment of particles. Possible improvements include the trimming of the cell arrays after the reassignment step in order to avoid gaps between the particles and to decrease the number of particles that are checked, as soon as an invalid index is encountered no further particles in that cell need to be checked. Compare cell c1 and cell c4 in figure 3.5, where this can be seen. In the current implementation always all particles in the cell are checked.

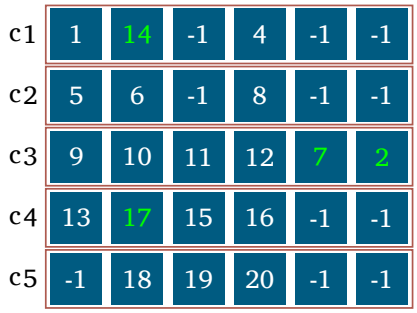


Figure 3.5: Green particles represent new particles located in cells after *assignParticles*() kernel.

4 Results

We chose two fundamental parameters for benchmarking, the grid size and the number of particles per cell. Various types of runs are plotted to compare algorithms and architectures.

The physical domain is a three-dimensional cube, which is divided into identical hexahedron elements.

Due to the chosen parameters the total number of particles differs. While the average number of particles per cell is kept constant and set to 10, the number of grid cells per dimension is varied from 10 to 100 thereby changing the number of particles in the system between 10^4 and 10^7. Results for time measurements are shown in Figure 4.1.

Measurements are obtained from the supercomputer JUROPA3. This supercomputer is equipped with differing types of compute nodes: (i) two Intel Xeon E5-2650 CPUs and two NVIDIA Tesla K20X GPUs and (ii) two Intel Xeon E5-2650 CPUs and two Intel Xeon Phi 5110P co-processors. This hardware is referred to as CPU, GPU and MIC in this report [1, 2, 7].

To study the behavior of more complex computations we implemented the force calculations between particles in line with a Lennard Jones potential [4, 9] for both algorithms. In order to compute the force acting on one particle mutual distances between the tagged particle i and all other particles j located in the local and the 26 neighbor cells have to be taken into account. Each work-item calculates a partial force contribution $\boldsymbol{f}_{ij}$ to the total force $\boldsymbol{F}_i$.

The GPU is the fastest architecture for force calculation and sorting where the CPU is the slowest as expected. Regarding the container-based approach the CPU-based version is the fastest because in this algorithm, many if-else statements are used inside the code to check for invalid particle indices.

After many time-steps, the particlesInCells array gets scattered by placeholder indices which are left where particles moved to a different cell. Further analysis and benchmarks are required to comment on an unexpected behavior of accelerators observed here for this algorithm. The minor performance of this implementation is probably due to the fact that there are many placeholder indices'-1' to be checked.

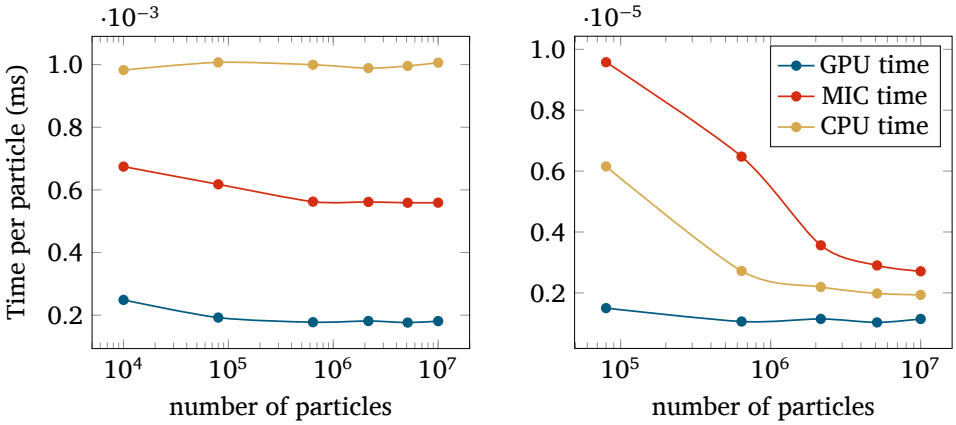


Figure 4.1: Comparison of GPU and MIC – linked-cell list approach. Figure 4.1a shows the runtime for the force calculation, figure 4.1b shows the runtime for linked-list cell creation.

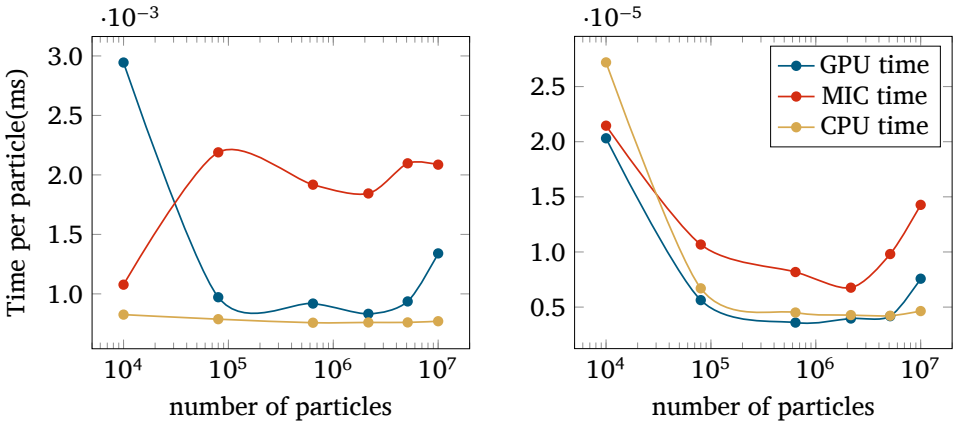


Figure 4.2: Comparison of GPU and MIC – container-based approach. CPU shows expected behavior with constant elapsing time where GPU and MIC shows non-constant behavior due to the number of particles. Figure 4.2a shows the runtime of the force calulation, 4.2b shows the runtime of sorting and administration.

5 Conclusion & Outlook

The container-based algorithm provides three-dimensional thread ranging where linked-cell list runs is one-dimensional. Since we have cells across work-items there is no perfect load balancing as one cell may have more particles than the other cells. In the linked-cell approach there is a perfect load-balancing because particles are distributed equally across work-items.

Optimization of the code has not been done so far. The container-based approach is more appropriate to make use of the fast local memory on the device. On the other hand, in order to treat the placeholder indices'-1' there is a lot of if-else branches. Moreover vectorization optimization has not been exploited in the code which would boost MIC performance closer to GPU since MICs are specially developed for vectorization.

However, the container-based algorithm is relatively slower than the linked-cell list algorithm since it has redundant '-1' ones in the array and worse memory locality after many time-steps. Moreover, there is the flag checking for every item of the `particlesInCells` array. In order to avoid this a new data-storage model could be developed. Here some important optimization issues to consider:

OpenCL devices are so different (CPUs, GPUs, and MICs) portable performance is a challenge.

1. efficient access to memory, i.e. memory coalescing and memory alignment
   - try to provide data to the work-items in parallel, e.g. Structure of Arrays (SoA) suits memory coalescence on GPUs and Array of Structures (AoS) may suit cache hierarchies on CPUs
2. vary number of work-items and work-group sizes due to architecture and the application
   - try to provide sufficient parallelism for the work-items; occupancy is a measure of how active each processing-element is kept
3. avoid work-item divergence:
   - try to avoid branches in device code; here barrier functions are helpful

So far, only one aspect regarding memory optimisation has been considered. A speed-up of 1.3 could be achieved by re-arranging the data organisation in the linked-cell list creation, e.g. splitting the struct array into float and integer arrays.

6 Acknowledgements

Firstly, I would like to express my deepest gratitude to my supervisors Dr. Godehard Sutmann, Willi Homberg, Rene Halver for sharing their experiences and support during the programme. Moreover, I would like to thank our to great organisator of the programme Ivo Kabadshow for everything he has done for us. There was nothing absent but more than I expected. For the last and the best, I want to thank my colleagues for a great time having with them while working on this project.

REFERENCES

[1] **Intel Xeon Multicore Processor E5-2650**. http://ark.intel.com/de/products/64590/Intel-Xeon-Processor-E5-2650-20M-Cache-2_00-GHz-8_00-GTs-Intel-QPI/.

[2] **NVIDIA Tesla GPUs for Server and Workstations**. http://www.nvidia.de/object/tesla-supercomputer-workstations-de.html.

[3] R. Halver and G. Sutmann. **Multi-threaded construction of neighbour lists for particle systems in openmp**. In *Parallel processing and Applied Mathematics, to be published*.

[4] J. E. Jones. **On the determination of molecular fields. ii. from the equation of state of a gas**. *In Proceedings of the Royal Society of London A:Mathematical, Physical and Engineering Sciences*, 106:463–477, 1924.

[5] M. Jongmanns. **Combining Linked-Cell Lists and Verlet Lists in OpenMP**. 2015.

[6] **Khronos OpenCL**. https://www.khronos.org/opencl/.

[7] **Intel Xeon Phi Coprocessor 5110P (8GB, 1.053 GHz, 60 core) specifications**. http://ark.intel.com/de/products/71992/Intel-Xeon-Phi-Coprocessor-5110P-8GB-1_053-GHz-60-core.

[8] G. Sutmann and V. Stegailov. **Optimization of neighbor list techniques in liquid matter simulations**. *Journal of Molecular Liquids*, 125(2):197–203, 2006.

[9] G. Sutmann. **Molecular Dynamics – Extending the scale from Microscopic to Mesoscopic in Multiscale Simulations Methods in Molecular Sciences**. *NIC Lecture Series*, 42:1–50, 2009.

Brain Cortex Segmentation using Deep Learning

Monika Bajcer
Department of Mathematics
University of Zagreb
Croatia
mbajcer92@gmail.com

Abstract *Deep learning is a new field in Machine Learning that uses algorithms that are able to create data abstraction models. One of the most important tools are artificial neural networks that have been inspired by the attempt to mimic biological neural networks. Our goal is to train a computer to detect gray matter in a human brain and to do so, build a fully convolutional neural network. We have been inspired by GoogLeNet to built an inception model of a neural network to segment brain images.*

1 Introduction

The human brain is the main organ of the human central nervous system – the big boss of the humans body that runs the whole show. The cerebral cortex is the dominant feature of the human brain associated with higher brain function such as thought and action. It contains 86 billion nerve cells, which we call gray matter which is the object of our study. Our data consists out of 7404 images of human brain slices. Associated to it is a binary image – so called labels. Every pixel of a label is either one – if a corresponding pixel of a brain image represents a gray matter, or zero – if it does not. Examples of a brain image and its labels are depicted in figure 6.2.

Our goal is to train a computer with deep learning methods to distinguish the gray matter from the remainder parts of the brain. The idea is to present numerous examples to the machine so that it learns and abstract from them. Every picture has a high resolution of 5711×6572 pixels and if we consider the total 7404 brain images and its additional 7404 labels, we have a problem of respectable size, approximately one terabyte. Out tool in solving this problem are neural networks. The neural networks with at least one hidden layer are a strong tool that can represent any continuous function. In other words, it holds that given any continuous function $f(x)$ and some $\varepsilon > 0$, there exists a neural network $g(x)$ with one hidden layer such that $\forall x, |f(x) - g(x)| < \varepsilon$. Even though models with just one layer can represent any function, empirical observation have shown, that many layers converge faster towards the above inequality. We try to find good solution with different neural network models. Each model is trained for a day and supervised with the help of a loss function allowing us to observe the learning progress. Additionally, we also investigate how different hyperparameters change the results.

(a) Brain image (b) Labels

Figure 1.1: Slice of a human brain and labels marking its gray matter.

2 Background

Described above is a common problem called classification. It is the problem of identifying to which of a set of categories a new observation belong, based on a training set whose category membership is known. There are various approaches, such as decision trees, support vector machines or linear classifiers [2]. Out of these, we have selected neural networks for our attempt due to their great success in image processing tasks. The idea is to try out with a model called neural network inspired by biological neural networks which can approximate functions that can depend on a large number of input variables. The basic unit of a brain is a neuron. Each neuron receives input signals from its dendrites and produces output signals that it sends along the axon to other neurons. Figure 2.1 shows an example.

In a model of artificial neuron, inspired by biological neuron, the signals (x_i) interact multiplicatively with dendrites, which are also called weights (w_i). The idea is that weights are learnable and control the influence of one neuron towards another. The cell body sums the partial results up and applies a function f – the activation function – that transforms a set of input signals into an output signal. The task of an activation function is break the linearity of a neural network, allowing it to learn more complex functions. They are also important for squashing the unbounded linearly weighted sum from neurons, hence avoid large values accumulating high up the processing hierarchy.

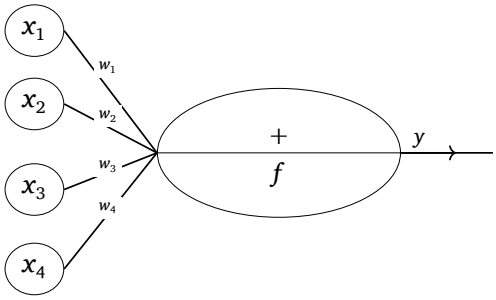


Figure 2.1: Mathematical model of a neuron – basic unit of a neural network.

There are many choices for an activation function and every one of them takes a single number and performs a certain mathematical operation on it. Some of the activation functions are:

- Sigmoid non-linearity has the form $\sigma = \frac{1}{1+e^{-x}}$. It takes a real number and gives an output in range between zero and one so large negative numbers will become close to zero and large positive numbers close to one. The sigmoid function has grown out of popularity, because it has two major drawbacks. Firstly, when neuron's activation is at either tail of zero or one, the gradient at these regions is almost zero and consequently, all learning stops in this section of the network. Beside that, sigmoid outputs are not zero-centered and this could introduce undesirable dynamics in the gradient updates for the weights.

- The tanh function squashes a real-valued number to the range $[-1, 1]$ and the output is zero-centered. It also holds that $\tanh(x) = 2\sigma(2x) - 1$.

- ReLu is short for the Rectified Linear Unit, a function that is becoming really popular because it will have fixed gradient (either zero or one, depending on the sign of x), it is numerically stable and fast to compute, easy to implement. Small difficulty for $x = 0$, as there is no gradient, usually randomly select one side (either zero or one). Its mathematical form is $f(x) = \max(0, x)$.

- Linear function where the weighted sum input of the neuron becomes the system output, also known as identity.

It is important to stress out that this model of a biological neural network is significantly simplified compared to actual neurons. One has to be prepared to hear groaning sound from anyone with some neuroscience background, if you draw analogies between neural networks and real brains [9].

Neural networks are modeled as collections of neurons described above, that are connected in an acyclic graph so that outputs of some neurons can become inputs to other neurons. A feed forward network is organized into distinct layers, shown in figure 2.2 in which neurons between two adjacent layers are fully pairwise connected. Neurons in the same layer are usually not connected. This kind of neuron organization is called a fully-connected layer.

The structure of neural networks makes it very simple to evaluate this networks using matrix vector operations. All weights for one layer can be stored in a single matrix so with a few multiplications neurons in the activation layer are evaluated. We will try out networks that will be made up of ten to 20 layers. The large number of layers is the reason, we denote it deep learning.

Large neural networks can represent more complicated functions but with more layers it is easier to overfit the training data. In other words, model memorizes the training data instead of abstracting it. To avoid overfitting we include special layers into our model, e.g. dropout or pooling layer. In practice, it shows that it is always better to use some methods to control overfitting instead of decreasing the number of layers and neurons because the smaller networks are harder to train.

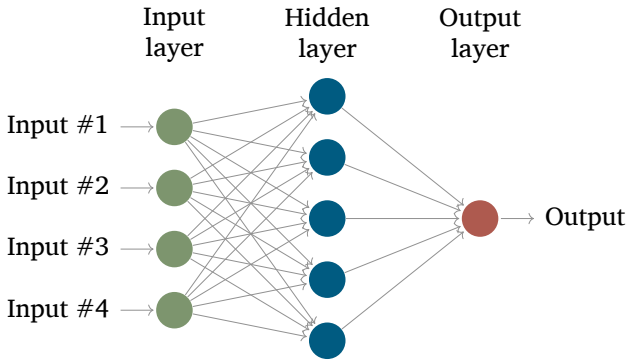


Figure 2.2: A model of a simple neural network with two layers.

The training of a model will be supervised with the help of a loss function. A loss function is a function that measures cost or deviation from the expected outcome (labels) compared to the models prediction. There are various loss functions in use. The most common for neural networks is mean-squared error (MSE). It takes an average over the data losses for every individual example, so $L = \frac{1}{N}\sum_i L_i$, where N is the number of training data and

$$L_i = \|f_j - f_{y_i}\|_2^2,$$

where f_j is $j-$th prediction and f_{y_i} is $i-$th ground truth label. The reason the $L2$ norm is squared is that the gradient becomes mush simpler, without changing the optimal parameters since squaring is a monotonic operation.

At this point neural network becomes an optimization problem that tries to find the set of parameters that minimize the loss function. Since the minima cannot be analytically determined, we do iterative refinements, where we start with a random set of weights and refine them. We compute the direction of negative gradient along which we change our weight vector because it is mathematically guaranteed to be the direction of the descend. Considering the data volume, we use a stohastic gradient descent (SGD). It is a way of computing the gradient over batches of the training set in order to perform a parameter update. The size of the mini-batch is a hyperparameter that we have to tune and it shows that the gradient from a mini-batch is a good approximation of the gradient of the full data. We have a special way of computing gradients of expressions using chain rule for partial gradients. Through iterative minimization of the loss and the backpropagation [5] of the gradient changes on the weights, the network learns from the data how to approximate the underlying problem function. These iterations we call epochs so when we say that the model did training in 1000 epoch, we mean that the weights had been refined 1000 times.

3 Convolutional Neural Networks

We will try out modern convolutional neural networks that are very similar to ordinary neural networks – they are organized in layers made up of neurons that receive inputs,

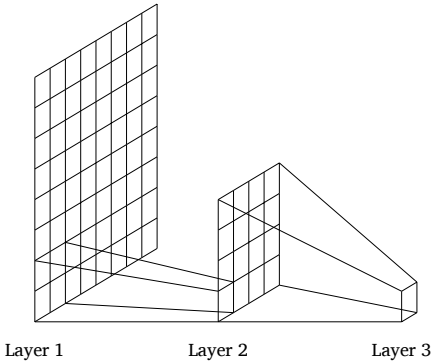


Figure 3.1: Convolutional layer neurons are only connected to the input's local regions.

performs a dot product and applies activation function. A convolutional neural network (CNN) allows the input to be an image or in general, any n-dimensional input. The neurons in a layer are only connected to a small region of the layer before it, instead of being fully pairwise connected with the neuron of the adjacent layer.

3.1 Types of layers

3.1.1 Convolutional layer

The convolutional layer is the basic layer for those kind of networks. This layer consist of a set of learnable filters that perform some operations only on a small portion of input image. We will work with images which are two-dimensional. Filter on a first layer of a convolutional network might have size 5×5 (five pixels width and five pixels height). On a figure 3.1 there is an example of sequetional convolutional layers. We then slide this filter across the width and height of the input and compute dot products between the entries of the filter and the input at any position. On a intuitive level, network should learn filters that activate when they see some type of visual feature and we will have a set of filters. In other words, filters look at only a small part of the input and share parameters with all neurons to the left and right spatially in order to save some sort of memory. This approach is practical because of local connectivity and sharing parameters, reducing the number of total parameters by an order of magnitude. Consider a network like in a figure 2.2 with an input like ours – an image of 5711×6572 pixels – it would take a lot of memory and it would also be slow to train.

One can pad the input with zeros around the border, which allows us control the spatial size of the output.In general, padding must be with a neutral element or number which in our case is zero. For a input size W, the filter size F, the stride S and the amount of zero padding used on the border P, we can compute the output size like $\frac{W-F+2P}{S} + 1$. It turns out that we can also reduce the number of parameters if we assume that if one feature is useful to compute at some spatial position (x_1, y_1), then it should also be useful to compute at a different position (x_2, y_2). In other words, learned features should be translation-invariant and should be able to detect in an arbitrary data position. That is why it makes sense to constrain the neurons to use the same weights. On figure 8.1 there is an image of two position over the input, and shows that each element is computed by

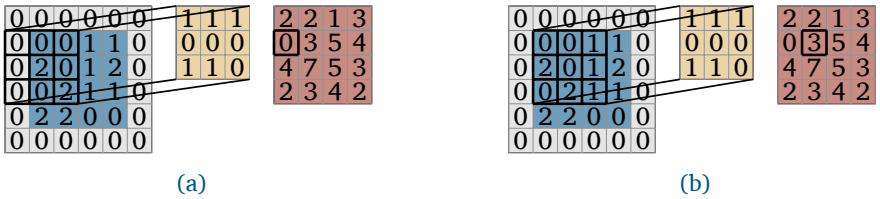


Figure 3.2: Two iterations in computing output of a convolutional layer, where blue is the image, gray zero padding, yellow are weights and red is the next layer.

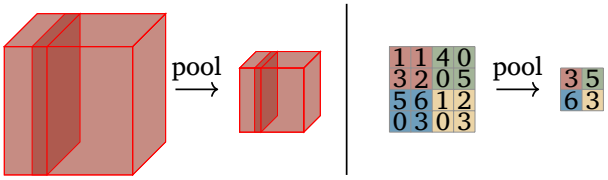


Figure 3.3: Pooling layer downsamples the volume spatially, independently in each depth slice of the input. Presented pooling method is a max pooling (selects largest number).

elementwise multiplying the input with the filter and summing it up.

3.1.2 Pooling layer

In our model we use periodically pooling layer in-between convolutional layers to progressively reduce spatial size, hence reduce the amount of parameters and computation in the network and also control overfitting. For example, if we use form of a pooling layer with filters of size 2×2 and a stride of two, as a result, we downsample the input by two along both width and height. In figure 3.3 there is an example of how a pooling layer is applied to every slice in a three-dimensional input. In our case the third dimension is one. In other words, we have images which are a two dimensional input. In this case, 75 % of the activations are being discard. With this layers we have to be careful because larger receptive fields can be too destructive.

3.1.3 Dropout layer

Dropout layer is an effective and simple layer that helps us provide overfitting [7]. It is implemented by only keeping a neuron active with some probability p, otherwise a neuron is set to zero.

4 Implementation

We do the training on a supercomputer Jureca [3] using one graphic card and 4 tasks, with every task doing the training with different learning rate. The implementation has been done in Python using Keras [4] library and the data has been saved as HDF5 file [1]. We also made a generator that takes a given number (in out example, two) of small random portions of images. Those portions we call windows and we estimate the window size to be 100×100 pixels. In the first example of convolutional networks, the output was always one pixeland later, in section 6 we change it to predict more pixel at once.

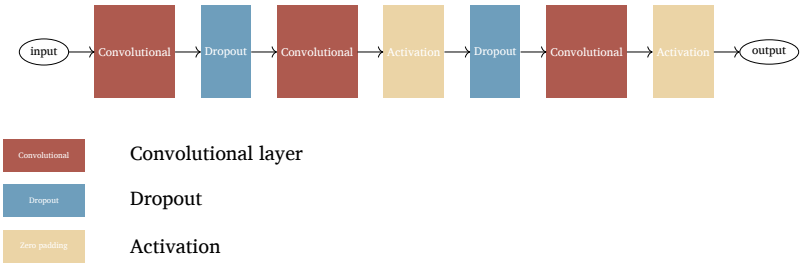


Figure 5.1: First model of a convolutional neural network with seven layers.

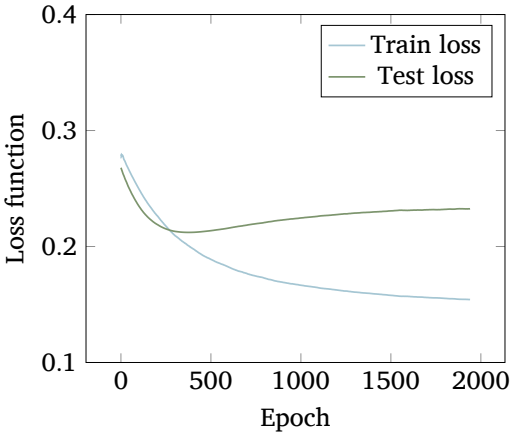


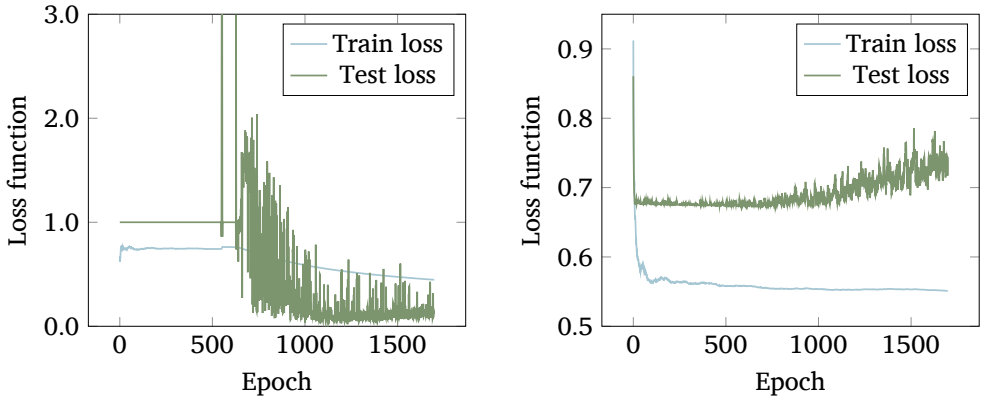
Figure 5.2: Example of overfitting – the train loss decreases, the test loss increases.

5 First Convolutional Networks and results

After having a look into the theory behind neural networks, we are now ready to construct a first simple neural network and to try it out to see if it predicts what we want. In figure 5.1 is a graph that shows our first network's structure. It consists of convolutional layers followed by dropout layers to prevent overfitting. The intermediate activation layers are ReLu units, whereas the final output is linear. The model is implemented in Python using Keras library. We split the data into the training set (80 %) and the test set (20 %).

We transformed labels to be minus one and one instead of zero and one. That way, when we set the last activation function to be linear, wrong predictions of zero would be more punished. We did that because model tend to predict more zeros then it should. Since we did that and since we put the activation function to be linear, if the final output will be close to minus one, we will assume that is not gray matter and for predictions close to one, we assume that the prediction is a gray matter.

The training usually lasted for a day because that was our computational limit on Jureca, even though the training could last longer. This was really expensive and so we had to be careful not to train models that will most likely not be good enough. The data loss takes the form of an average over the data losses for every individual example. The



(a) Learning rate = 0.001

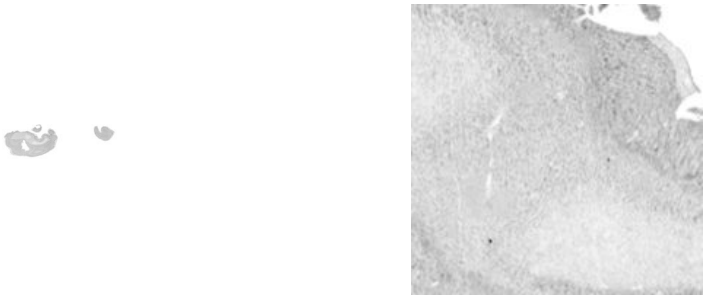
(b) Learning rate = 0.00001

Figure 5.3: Two examples of plots of loss function with different learning rates.

goal is to make the loss function to be really small because that means that predictions are good. The training consisted of more then 1000 epochs, and after every epoch, we plotted the loss function, both data loss and train loss. The reason is that the plot will that way tell us not only about the loss, but also about when overfitting accures. When we see that the train loss started to decrease, but test loss increase, like in a figure 5.2, it was a sign that a model is just memorizing the data and does no abstractions. On this plot it is also visible how fast the model learns and speed depended on a learning rate. Small learning rates are likely to lead to consistent but slow progress. Large steps can lead to better progress but are more risky because we could skip the minimum of a loss function. So the step size became one of the most important hyperparameters that had to be carefully tuned. That is why we will try out different rates and then see which one works best. We did the training on Jureca [3], using one node and four tasks per node, with every task doing the training with different learning rate using systematic parameter search. The training process is visible on plots. Two examples of plots are on figure 5.3.

After a day of training, we want to test our model. When we gave it a picture, it will make prediction pixel by pixel, and for every pixel it will tell us if it is a gray matter or not. First problem in that idea was the size of the picture. Let's repeat, the pictures were 5711×6572 big and it would take a lot of time to make predictions for every pixel in one picture in a day. So that is why we tested it on a small part of one picture, like in figure 5.4.

On figure 5.5 are our first results – it is obvious that they are not good enough. That is why we had to go for something better.



(a) How a full brain image looks like. (b) Image portion to predict.

Figure 5.4: Testing model how it predicts small portion of a brain image.

(a) Ground truth (b) Our results

Figure 5.5: Testing a model how it predicts small portion of a brain image – label and results.

6 Going for deeper ConvNets and new results

6.1 Inspiration in Google

Unhappiness with our old results made us think about other possibilities for constructing a CNN model. This time, we found inspiration in GoogLeNet, [8]. What is so special about this architecture are inception modules – this is a convolutional network that has smaller convolutional networks build inside, eliminating a large amount of parameters that do not seem to matter much. The model that we made, shown on figure 6.1, consist of many layers, mostly convolutional, followed by pooling layer or dropout in order to prevent overfitting.

Another thing that was now changed was number of output pixels, so instead of predicting one pixel at a time, we went fully convolutional and now are able to operate on any input size and produce an output of corresponding size (eg. 6×6 or 24×24 output size). An illustration of this is on the figure 6.2. We put the same problem in front this model, like in the figure 5.4. The results were again not good, see figure 6.3.

6.2 Removing pooling layers

Now we started to think about why our model was not able to learn anything and why it gave us sub-optimal results. To prevent overfitting, we used many pooling layers and with that we did not allow the model to learn. Instead of control overfitting, we went to other extreme – underfitting, where model is not able to generalize, so the new idea was to remove most of the pooling layers and to challenge a model one more time. We now have a model on a figure 6.4a, we train it for a day with different learning rates in order to compare and to choose a learning rate that gives us the smallest loss function. New prediction of a small image portion from figure 5.4 is on figure 6.4b. The model still does some errors, but now it recognized the structure of a gray matter in a brain and results make much more sense.

6.3 Results on full images

After we got better results on a small portion of an image, we are now ready to make predictions on a full image. Our input was of a size 100×100 pixels and corresponding output was a size 44×44. The results of a prediction on a whole picture are on figure 6.5. Beside that, we gave him more images to make a predictions, which are shown in a figure 6.3.

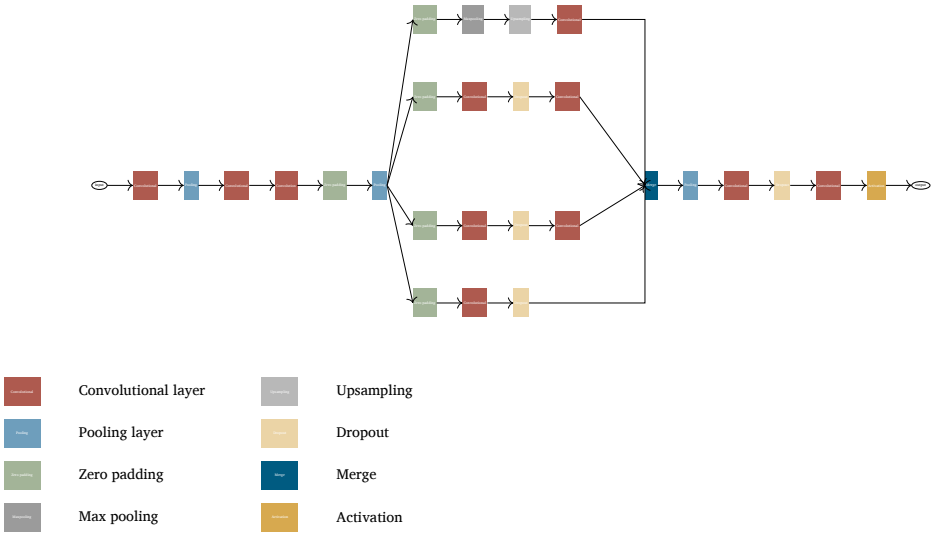


Figure 6.1: New model of CNN inpired by GoogLeNet with inception module.

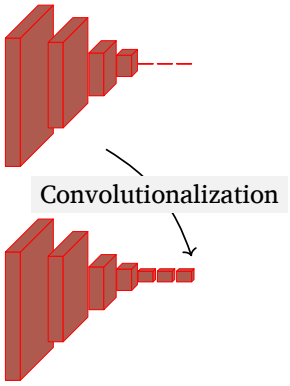
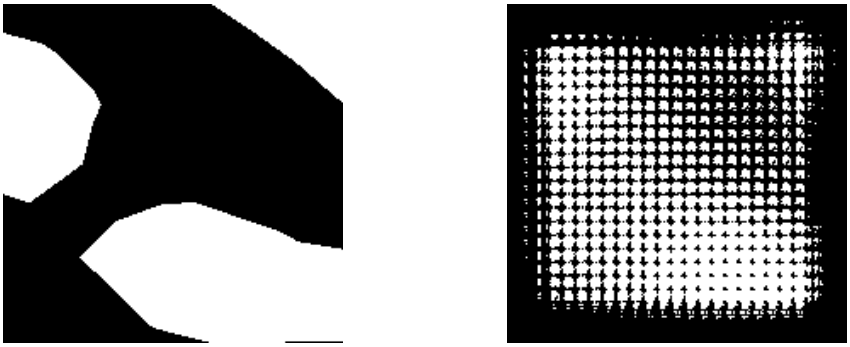
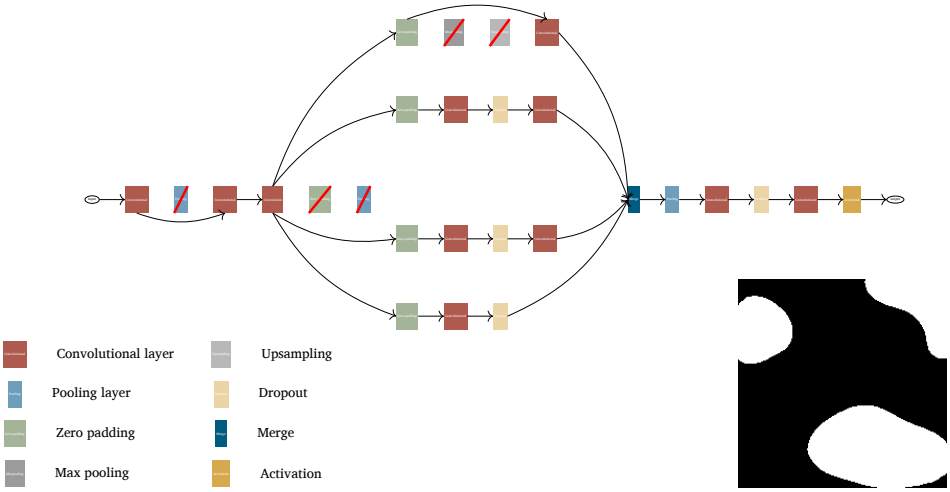


Figure 6.2: Ilustration of transforming a model into fully convolutional.



(a) Ground truth (b) Our results

Figure 6.3: Result that we got from model on a Figure (6.1).



(a) Removing pooling layers (b) New results

Figure 6.4: Transforming our model and allowing it to learn more.

(a) Brain image (b) Label (c) Results – treshold 0.1

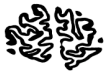
Figure 6.5: Predicting the whole image, 44 × 44 pixels by one.

(a) Brain image

(b) Label

(c) Results

(d) Brain image



(e) Label

(f) Results

Figure 6.6: More predictions of the whole image, 44 × 44 pixels by one.

7 Discussion

When we want to determine a gray matter of a brain, we give our model an image like on figure 7.1a and it gives us predictions like on figure 7.1c, this is how it looks like without any post processing. What we want is an binary output, with zeros on the places that are under a given number, and ones on the places that are above. This number that makes the difference we call a treshold. In the results in section 6 we tried out different tresholds and we show the one that gives the best results. On figures 7.1, 7.2, 7.3 we show three examples how different tresholds influence results on different sections of a brain (on figure 7.1f there is an example where all the predictions are smaller then the treshold). It still has to be investigated what is the proper post processing procedure which would give us best results. In other words, one could determine the treshold function that should be applied to the results of our model.

(a) Brain image (b) Label (c) Result

(d) Treshold: -0.35 (e) Treshold: -0.2 (f) Treshold: 0.35

Figure 7.1: Enlarged brain image and results.

(a) Brain image (b) Label (c) Result

(d) Treshold: -0.35 (e) Treshold: -0.2 (f) Treshold: 0.35

Figure 7.2: Brain image and results.

(a) Brain image (b) Label (c) Result

(d) Treshold: -0.35 (e) Treshold: -0.2 (f) Treshold: 0.35

Figure 7.3: Enlarged brain image and results.

8 Summary and Outlook

Fully convolutional networks are a rich class of models, of which we tried out a model inspired by a GoogleNet. In this paper, we achieved some results on the brain cortex segmentation and now our model can distinguish gray matter from the rest part of a human brain. This model still makes mistakes sometimes and there is a place to improve it. This project was just a small puzzle inspired by the big Human Brain Project [6] and next step would be to make three-dimensional model (predict multiple images at once) and get a deeper look into the brain.

9 Acknowledgements

I would like to thank my supervisor Dr.-Ing. Morris Riedel for giving my the opportunity to be the part of this programme. I also owe my gratitude to Christian Bodenstein and Markus Götz for their help and guidance. Also, I truly appreciate Dr. Ivo Kabadshow for his great organisation, for always being there for all the gueststudents, and especially for his help with the presentation. Last, but not least, I thank my fellow guest students for making this programme a special experience that I will never forget.

References

[1] **Hdf5 for python**. `http://www.h5py.org/`.

[2] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*. The Morgan Kaufmann series in data management systems. Elsevier, 2006. `https://books.google.de/books?id=eKJvngEACAAJ`.

[3] **Jureca user info**, 2016. `http://www.fz-juelich.de/ias/jsc/EN/Expertise/Supercomputers/JURECA/UserInfo/UserInfo_node.html`.

[4] **Keras documentation**. `https://keras.io/`.

[5] M. Nielsen. **How the backpropagation algorithm works**, 2016. http://neuralnetworksanddeeplearning.com/chap2.html.

[6] **Human brain project**. https://www.humanbrainproject.eu/.

[7] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. **Dropout: a simple way to prevent neural networks from overfitting.** *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[8] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. **Going deeper with convolutions**. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[9] **Cs231n convolutional neural networks for visual recognition**. http://cs231n.github.io/.

Brain simulators on JULIA: Initial performance evaluation

Patrick Emonts
Jülich Supercomputing Centre
RWTH Aachen University
Germany
patrick.emonts@rwth-aachen.de

Abstract *In this report, we describe an initial performance analysis of the prototype Cluster JULIA installed in the framework of the Human Brain Project (HBP). Special care is taken to understand the memory hierarchy of the Intel Xeon Phi Knights Landing processor. Finally, the performance of the neural simulator NEST is evaluated.*

1 Introduction

Understanding the human brain is one of the big visions of mankind: Revealing the processes behind reasoning and creativity, building artificial intelligence. The Human Brain Project, funded by the European Union, unifies these efforts into one project.

One idea is to simulate the brain on high-performance computers in order test and refine models. An example for one such simulator is NEST which builds physiological inspired neural networks. These networks consist, just as the brain, of neurons and connections between neurons. Currently, networks up to 1.73 billion neurons can be simulated on the biggest computers using up to 1 PB of main memory [5]. In contrast, the human brain contains up to 100 billion neurons and even more synapses. Trying to reach the human brain in size and complexity, new computer architectures have to be used.

The newly released Intel Xeon Phi Knights Landing (KNL) platform is a promising candidate in this regard. Benefiting from up to 72 cores per card, codes can be executed in a highly parallel fashion on a single processor. Before exploiting the KNLs for neuroscientific applications, the actual performance of the processors is analysed with micro-benchmarks.

The primary target of this study is to provide a first understanding of the memory hierarchy and to give an idea of neural simulations on KNL. In order to understand the hierarchy, a memory micro-benchmark is used. The tool of choice was PMBW which implements read and write operations in a multi-threaded way [7]. It resolves unique features like highspeed memory access to multi-channel DRAM (MCDRAM), which is located directly on-package. Additionally, an initial benchmark of the Solid State Disks (SSD) on the DataWarp nodes is performed. These disks serve as an additional storage layer between the fast RAM and the slower network storage. The second step consists of timing runs for NEST examining the scalability and time to solution performance.

2 JULIA Cluster

The JULIA Cluster system is a high performance system built by Cray and based on the Intel Xeon Phi Knights Landing (KNL) technology. It consists of 2 Login-nodes, 60 KNL-, 4 DataWarp-, and 4 Visualization nodes. Each KNL node consists of a Intel Xeon Phi CPU 7230 that has 64 cores which support 4 hardware threads. The memory of the KNL-card is split in two parts: The fast MCDRAM (16 GB) and the slower but larger DDR4 RAM (96 GB). All other nodes are equipped with Intel Xeon CPU E5-2680 v4

operating at 2.4 GHz and 128 GB of DDR4 RAM. Two 14-core processors are hosted on a two socket system working on different NUMA (non-uniform memory access). They act as single node with 28 cores. This holds true for DataWarp-, Login-, and Visualization nodes. Only the DataWarp nodes are equipped with actual hard-drives (2 Intel NVMEs of 1.6 TB). In order to provide immediate visualization capability, 4 NVIDIA graphics card (GK110BGL) are attached to each visualization node.

2.1 Intel Xeon Phi Knights Landing Platform

The most salient feature of the JULIA cluster is the Intel Knights Landing chip, the latest Xeon Phi processor. In contrast to the Knights Corner coprocessor, the Knights Landing version can be shipped as a coprocessor or as a processor. This report only considers the Knights Landing processors since they are installed in the JULIA cluster.

2.1.1 Architecture Overview

The Knights Landing platform consists of mulitple architectural layers. An overview of the setup is provided in Figure 2.1. The elemental building block of the KNL processor

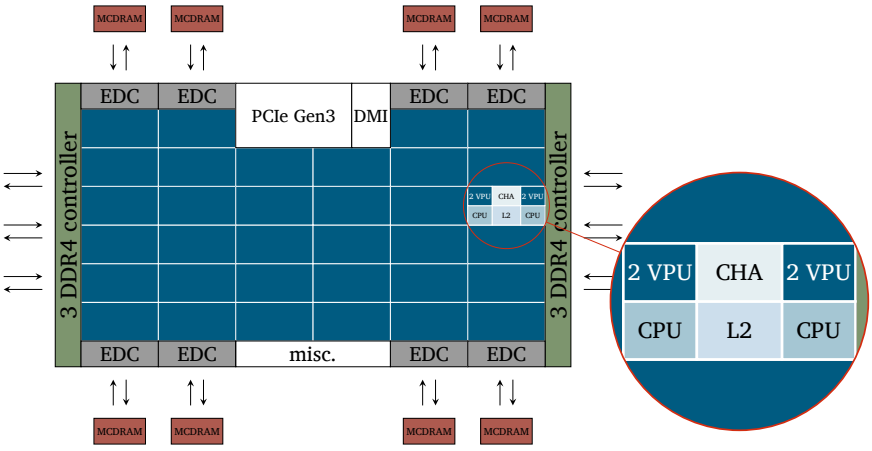


Figure 2.1: Schematic drawing of the Knights Landing architecture. The blue tiles in the center consist of two processors with two vector processing units (VPU) each. They can be observed in further detail in the inlay. The caching home agent (CHA) ensures cache-coherency across the tiles. Memory controllers for the DDR4 RAM are marked in green at either side of the card. The red boxes above and below represent the high-bandwidth MCDRAM.

is called tile (marked as dark-blue Figure 2.1). One tile consists of two modified Intel Atom cores (codename Silvermont) with two vector processing units (VPU) each. The two out-of-order cores share 1 MB of L2 cache, while using 32 kB of instruction L1 (IL1) and 32 kB of data L1 (DL1) cache.

In contrast to other processors, the KNL provides two layers of main memory: multi-channel DRAM (MCDRAM) and DDR4 RAM. The first is on-package memory and provides 16 GB of high-bandwidth memory with a transfer bandwidth of approximately 450 GB s^{-1} [4]. The DDR4 RAM is addressed via six memory controllers, three at each side of the KNL (marked in green in Figure 2.1). A maximum of 384 GB can be addressed and used with a theoretical bandwidth of 115 GB s^{-1}.

2.1.2 Memory and Cluster modes

The KNL cards provides serveral modes to exploit the high-bandwidth of the MCDRAM which have to be selected at boot-time. These modes are grouped in to sections: Memory and Cluster modes. The latter affect the cache look-up strategy while the first influence the structure of the MCDRAM. The user may choose freely from both sections. At first, the different Memory modes will be presented.

Cache Mode This modes uses the MCDRAM (the high-bandwidth memory) entirely as cache for the DDR memory. It is important to note that the cache does not work as an L3 on the processor side, but as a pure memory cache on the DDR side. Since the organization of the memory cache is not accessible from the software side, the MCDRAM remains invisible to the programmer. All allocations will be made directly in the DDR.

Flat Mode The Flat Mode renders the MCDRAM visible to the software. The high-bandwidth memory is attached to DDR4 address-space on the high-address side. It can be either allocated directly via software (memkind for C, FASTMEM for Fortran) or used as a preference for the whole program via numactl.

Hybrid Mode The Hybrid Mode splits the MCDRAM in two parts. One will be utilized as cache, the other can be addressed directly. Hence, it is a combination of the two modes mentioned above.

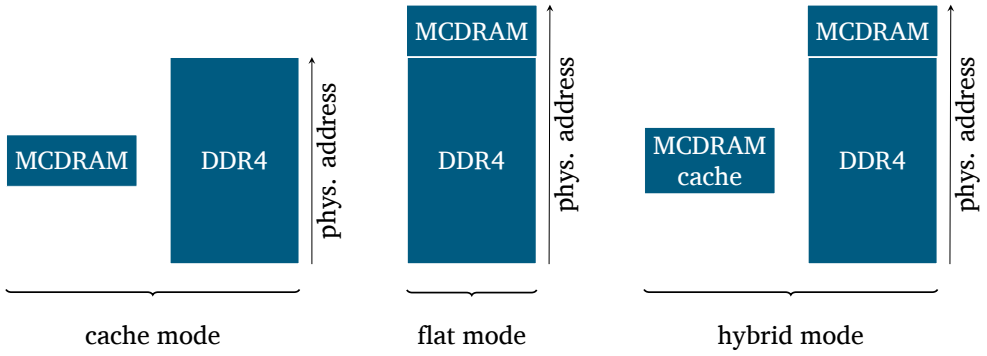


Figure 2.2: Illustration of different memory modes. In Flat and Hybrid Mode, the MCDRAM is accessible by software via the given addresses. MCDRAM cannot be accessed directly in Cache mode.

The different modes are illustrated in Figure 2.2.

Independently, one of five cluster modes can be selected: all-to-all, quadrant, hemisphere, or subNUMA-cluster-4 (SNC-4), or SNC-2. In this work, only the modes all-to-all and quadrant will be used. Further information may be found in literature [4]. The cluster mode influences the behaviour of the KNL upon the occurance of a cache miss. If a cache miss occurs, three agents are involved: tile, Caching-Home agent (CHA), memory. The affinity between these agents is managed by the selected Cluster mode.

The most general mode is the all-to-all mode. It is the only mode that works with different sizes of RAM attached to the different DIMMs. Since no assumptions are made about the layout of memory, it is, in general, expected to be the worst mode in terms of performance [4]. No affinity is assumed between the CHA, the tile, and the memory. Therefore, any tile may request any address of memory which is tracked by a tile anywhere on the chip. On average the messages have to transverse longer on the chip and the lookup for an L2 cahce miss will take longer. For further details the reader may be referred to Ref. [4].

The default mode for equal memory distribution is quadrant. In contrast to all-to-all, an affinity is established between memory and CHA. The CHA is divided into four parts and each part organizes the tags for the memory in its quadrant. Therefore, the expected latency of memory look-ups is smaller.

A third mode is SNC-4 which provides a strong affinity between all three agents. It is meant for NUMA-optimized software and can be most easily imagined as a multi-socket system. The KNL is divided into 4 subNUMA nodes and they can be readily depicted as a four-socket system built of XEON processors. The access-times for addresses that are stored in the memory of another socket are considerably higher.

For more information about other modes and further details the reader may be referred to literature [4].

3 PMBW Benchmark

The PMBW benchmark measures the memory bandwidth and memory latency for read and write operations. In order to avoid the dependence on compiler optimization each benchmark is written in inline assembly in a C-function. Often, a pure read and write operation without processing is regarded as unnecessary by the compiler and, therefore, does not occur in the final binary. This optimization would prevent the desired measurement and has to be avoided.

The functions are implemented using different SIMD[1] instruction sets. The KNLs support a maximal vector length of 512 bit which is exactly the length of one cache line. If the largest SIMD instructions are used, the full cache line is populated at once and the highest bandwidth is achieved.

PMBW is one of the few memory benchmarks that can measure the memory bandwidth while multiple threads are used. The implementation relies on the usage of pthreads. This feature is especially interesting since the KNL platform is expected to reach maximal performance for a large thread numbers. This expectation is motivated by the fact that one thread is not divisible and can only run on a single processor (one of 64) which leaves most of the card unused.

Due to limited timer resolution, each benchmark is executed for at least 1 s. Further details about the PMBW benchmark can be found online [7].

3.1 Performance with different instruction sets

In a first step, the different instruction sets are benchmarked one against each other on a single core. The double logarithmic plot in Figure 3.1 shows the result of a PMBW execution on a single thread. The names listed in the legend are always structured in

[1] **S**ingle **I**nstruction **M**ultiple **D**ata

the same way. *ScanRead* stands for the operation that is performed. It is a reading operation in a sequential manner, hence *Scan*. The following number names the size of the instruction operand in bits. The suffix *UnrollLoop* states that the 16 times unrolled version of the function was benchmarked instead of the simple implementation with one instruction per loop.

The constant offset of the different curves against each other shows the influence of the different instruction sets. Since the doubled amount of data can be transferred in an equal number of processor cycles, the observed bandwidth doubles as well (constant offset in the logarithmic plot). The x labels have been partially replaced by the actual sizes of important hardware features. For example, the bandwidth drops significantly when the arraysize exceeds the size of the L1 cache. The same behaviour can be observed for the shared L2 cache.

The absolute values of the benchmark can be compared to the hardware capabilities of the KNL. The KNL is able to perform 2 load instruction per cycle with a width of $64\,\mathrm{B} = 512\,\mathrm{bit}$ each. Running at a base frequency $f_{\mathrm{base}} = 1.3\,\mathrm{GHz}$, the nominal bandwidth can be calculated as $2 \cdot 64\,\mathrm{B} \cdot 1.3\,\mathrm{GHz} = 154.97\,\mathrm{GiB\,s^{-1}}$. The PMBW reaches in the L1 region up to $117.50\,\mathrm{GiB\,s^{-1}}$ which is reasonably close to the theoretical value.

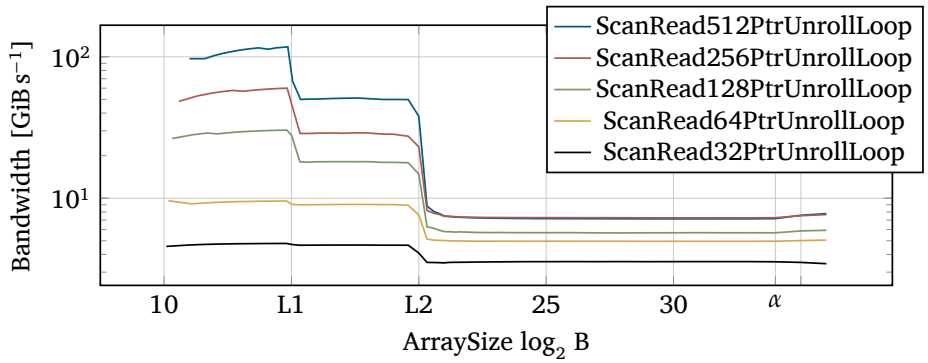


Figure 3.1: PMBW benchmark results for a KNL configured with Flat-Quadrant. The PMBW is executed on one thread and the instruction set is changed in each run. The label α denotes the size of the MCDRAM (2^{34} B).

3.2 Performance with multiple threads

The PMBW benchmark also provides the feature to measure the performance of one SIMD instruction set when used with a different number of threads (compare Figure 3.2). The overall shape of the graph is comparable to Figure 3.1. However, there are some interesting differences. These benchmark results do not only show sharp drops in performance when L1 and L2 are exceeded, but also when the capacity of MCDRAM is reached at 16 GiB. This drop only evolves for high thread numbers, implying a large number of threads is necessary to use up the whole bandwidth of the card.

Another difference is the right-shift of the different drops at cache size boundaries in contrast to the single threaded benchmark. Since the L1 cache is private to the processor, the total amount of L1 changes if multiple threads are in use. Therefore, the drop

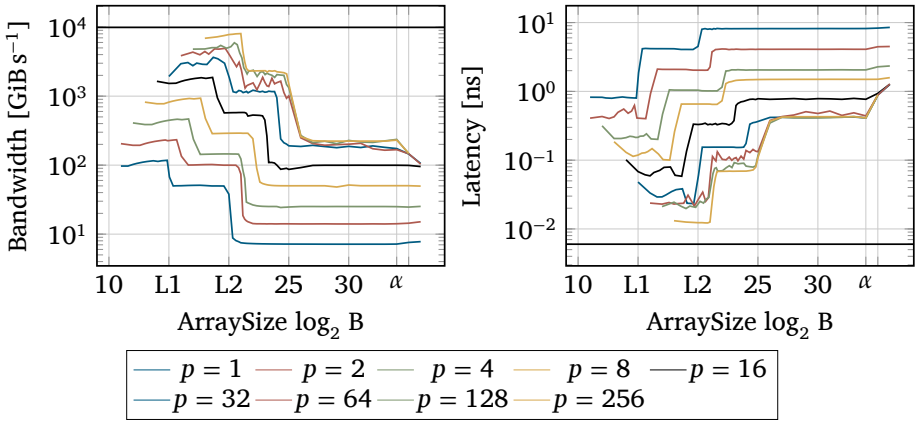


Figure 3.2: Left: Bandwidth measured with different numbers of threads. Right: Latency measured with different numbers of threads. In both plots, the actual array size has been replaced by sizes of hardware features. α replaces the size of the MCDRAM (16 GB). p defines the number of threads used in the benchmark. The solid black line denotes the theoretical limit.

to lower bandwidth values is observed at higher array sizes for increasing number of threads. A similar explanation is valid for L2 because it is shared inside a tile but not across tiles. Hence, the amount of L2 cache increases if more threads are used.

The theoretical maximum value that can be reached is a bandwidth of 9918.21 GiB s^{-1}. It is calculated as bandwidth $= n_{\text{load ports}} \cdot n_{\text{CPU}} \cdot f_{\text{base}}$, where $n_{\text{load ports}} = 2$ for the given processors used in the tiles.

The right plot of Figure 3.2 shows the latency plot corresponding to the measured bandwidth values. Here, the latency is defined as latency $= \frac{t}{n_{\text{accesss}}}$ where t is the runtime of the benchmark and n_{access} is the number of accesses of the memory during the run. The minimal latency for 512 bit operands amounts to 0.012 ns. This data is reasonable although it is smaller than the duration of one processor cycle (0.76 ns). The L1 cache is local to the processor and can be accessed by each of the 64 processors during each cycle. Thus, the minimal, theoretical latency is $\frac{1}{2 \cdot 1.3\,\text{GHz} \cdot 64} = 0.006$ ns, which is smaller than the smallest, measured latency value. The additional factor 2 is attributed to the fact that the processor can issue 2 instructions per cycle.

3.3 Comparison of different Cluster modes

As mentioned above, the all-to-all modes is the only one that manage differently sized RAM-blocks since it makes no assumptions at all. In contrast, the quadrant mode establishes a binding between the main memory and the CHA. Therefore, the load process from memory should be faster than in the case of all-to-all. However, Figure 3.3 shows a better performance for all-to-all in the MCDRAM region. The problem with this specific benchmark could be reproduced on another KNL system by Intel and is currently under investigation.

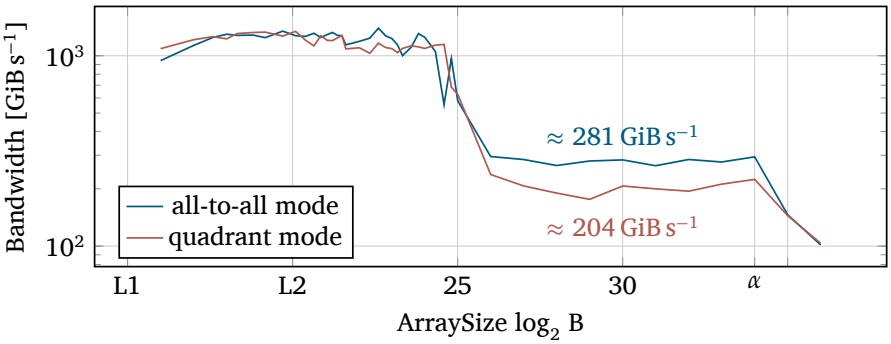


Figure 3.3: Comparison of PMBW running on 64 cores with AVX-512 instructions and different cluster modes. As above, α denotes the hardware size of the MCDRAM. The numbers inserted are the mean bandwidth values (harmonic mean) for allocation on the MCDRAM.

4 ezFIO

The only nodes that are actually equipped with non-volatile memory are the DataWarp nodes. The two attached NVMe (Non-Volatile Memory express) cards are Intel DC P3600 with 1.6 TB each. They are used as an additional hierarchy level of the permanent storage system.

If data is needed after the termination of a run, it has to be transferred to disk at some point of the program. Since the compute nodes do not have non-volatile memory attached and the network filesystem is too slow, an additional layer of durable memory is introduced: The DataWarp nodes. They work as a mediator between the high-bandwidth RAM and the inherently slower network filesystem.

In order to perform the measurements of the IO performance independent of a specific file system, ezFIO is used. It is a Python wrapper that uses FIO which accesses the SSD directly via libaio and the NVMe interface [8]. The data is not written to the filesystem which is mounted on the disk, but directly onto the device, wiping the memory in the process. The benchmark presented here considers only read and write operations to a single disk although the DataWarp nodes are equipped with two disks. The analysis of work sharing strategies between both disks could be part of a further analysis.

ezFIO does not show any significant alterations in the IO operations per second (IOPS) (see Figure 4.1). The bandwidth stays almost constant at around 165 000 IOPS indicating a constant bandwidth. Therefore, the NVMe's performance is not influenced by a wear leveling mechanism.

Additionally, ezFIO is used to check the ultimate read and write performance values which are given by Intel with 1600 MB s^{-1} for sequential writing and 2600 MB s^{-1} for sequential reads [3]. To test these numbers, the ezFIO was configured to perform sequential reads and writes on the bare disks. The results of these runs are presented in Figure 4.2. The benchmark was executed on one thread with increasing block size. Write benchmarks are executed with a queue depth of one while read benchmarks are executed with a queue depth of 256. Both benchmarks show that the limits reported by Intel are almost reached. The read benchmark reaches up to 2345.50 MB s^{-1} while the data is written with up to 1479.62 MB s^{-1}.

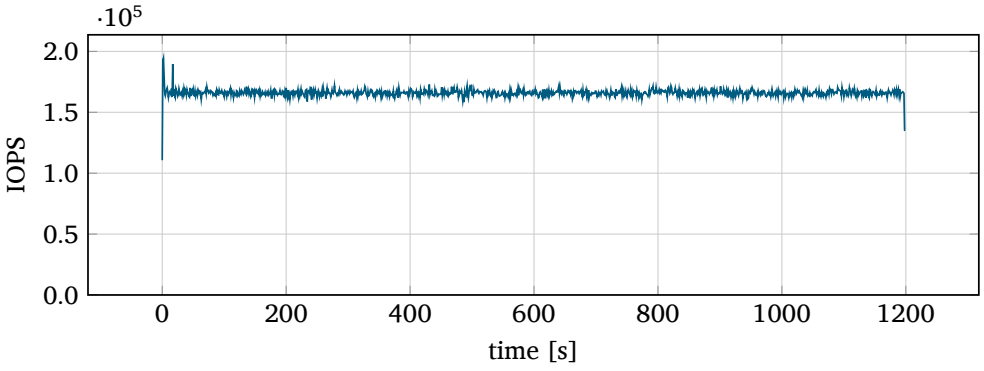


Figure 4.1: Bandwidth of the NVME over time. The bandwidth is monitored as number of IO operations per second (IOPS). It does not change significantly over time.

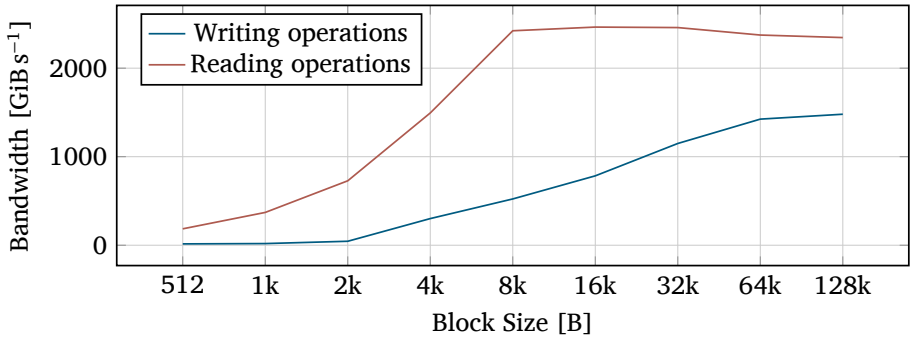


Figure 4.2: Sequential read and write operations with a single thread.

5 NEST – Network Simulator

The Neural Simulation Tool (NEST) is an application to simulate biological motivated neural networks with scalability in mind [1]. It offers a variety of neuron models as well as different synapse implementations. In this report, NEST was chosen as an application benchmark on JULIA. Therefore, the main focus is rather on performance of a single network than validating the actual results.

We chose the Brunel Network example implemented in Python as a benchmarking case (brunel_alpha_nest.py). The concept of a Brunel network is a neuron population that is only sparsely connected [2]. Given the order number n, the network consists of $N_E = 4n$ excitatory neurons and $N_I = n$ inhibitory neurons. Thus, 80 % of the population is excitatory and 20 % is inhibitory. Finally, each neuron receives $C_I = \varepsilon N_I$ connections from inhibitory neurons and $C_E = \varepsilon N_E$ from excitatory ones. Since the network is supposed to be only sparsely connected, ε is chosen as $\varepsilon \ll 1$.

In order to evaluate the performance of NEST on Knights Landing processors, both, the scaling with MPI and OpenMP is evaluated, since these strategies are used in the

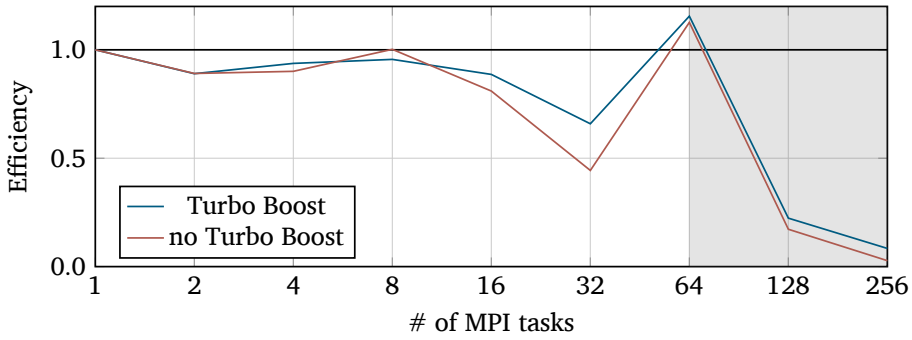


Figure 5.1: Strong scaling of NEST with varying number of MPI ranks. The number of OpenMP threads is held constant at 1. The shaded region indicates the SMT region of the KNL processor, i.e. the number of processes exceeds the number of cores available on the card.

implementation of NEST. The efficiency

$$\eta = \frac{t(1)}{n_{PU} \cdot t(n_{PU})}$$

is used to evaluate the scaling behaviour for a different number of PUs. Here, $t(n_{PU})$ is the execution time of the program on a n processing units (PU). A PU names a single thread, i.e. a single stream of instructions, but imply nothing about the affinity to a certain process. Therefore, 4 PU may be realized, for example, as 2 MPI ranks with 2 OpenMP threads each or as 1 MPI rank with 4 OpenMP threads.

In this report, we consider only the strong scaling performance of the program. Strong scaling is harder to achieve, since the total work stays constant during the analysis. Therefore, the work per processing unit will decrease as the number of PUs is increasing. Hence, communication and computation cannot be overlapped as efficiently and effects like synchronization may become visible.

The presented graphs are recorded with NEST 2.10.0 compiled with GCC 6.1.0 and linked against the Intel-MPI 5.1.3. The NEST software is not a benchmark on its own, but a scientific neural simulator. Therefore, NEST is not designed like PMBW or ezFIO to output data that is already a statistical value of multiple runs. The JUBE framework [6], written and maintained in Jülich, is used to organize multiple runs and to run a basic statistical analysis. Each data point is the minimum time of five independent runs. The minimum is chosen because the distribution of computing time is inherently asymmetric. It only has a fixed lower limit, while the upper limit may be, in the worst case, arbitrarily large due to operating system effects.

The efficiency plot for both cases, MPI and OpenMP, can be considered as fairly high. Especially the super-linear behaviour of the MPI version at full-use of the KNL processor looks very promising. Efficiency drops only for large a number of cores, when the SMT region of the KNl is reached, i.e. multiple threads are scheduled per core. The inferior performance of OpenMP with respect to MPI is expected, since the memory allocation of NEST is not yet optimized for high numbers of OpenMP threads. The optimization is planned for the upcoming releases.

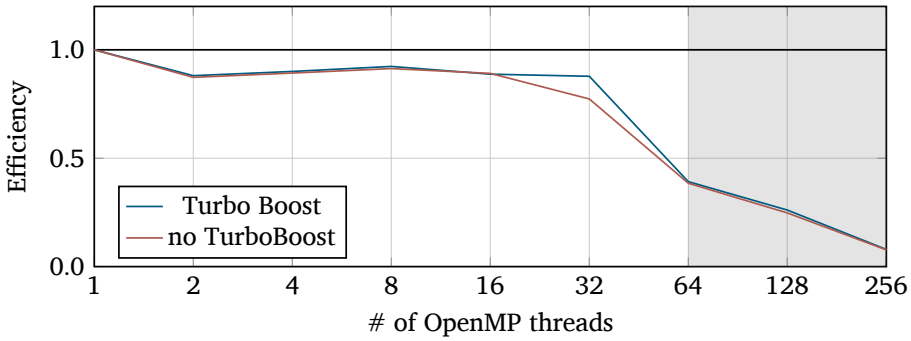


Figure 5.2: Strong scaling of NEST with varying number of OpenMP threads. The number of MPI ranks is held constant at 1. The shaded region is the SMT region of the KNL processor.

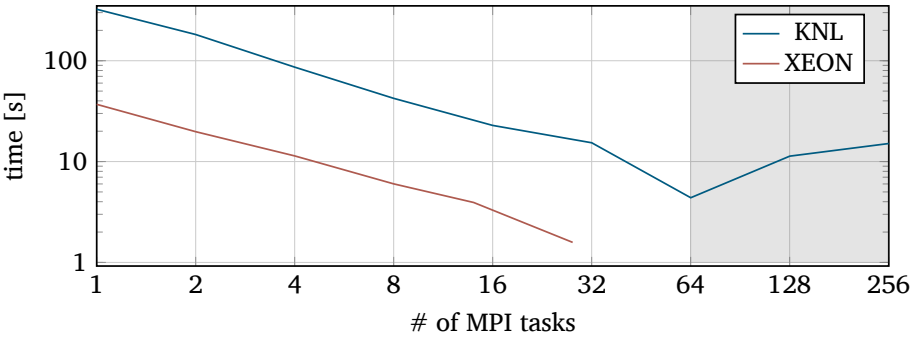


Figure 5.3: Comparison of KNL 7230 and Xeon CPU E5-2680v4 processor. The number of OpenMP threads is fixed to one during the whole benchmark. The shaded region highlights the SMT region, as above.

Although the efficiency of NEST on KNL platforms is very promising, the time to solution is slower than on Xeon processors.

The result shown in Figure 5.3 was produced using the visualization nodes of JULIA as a reference (compare Section 2 for further information on the hardware).

The reason for this performance difference is not yet fully clear. One explanation under investigation is the following: The Xeon Phi processor consists of a big number of low-performance core (Intel Atom). It is possible that the Xeon processor architecture is more suitable to the task than the massively parallelized approach with smaller cores because of a better processor architecture.

6 Conclusion

This report presented an initial performance analysis of JULIA's KNL- and DataWarp nodes. The hierarchical memory architecture of Intel Xeon Phi Knights Landing processors that incorporates an additional layer of high-bandwidth memory, can prove beneficial for programs that use ranges of data frequently which are larger than the processor's cache. The PMBW benchmark shows that maximal performance in terms

of memory bandwidth is only reached for high numbers of threads. An unexpected difference between quadrant and all-to-all Cluster mode is currently under investigation. Additionally, the SSDs used in the DataWarp nodes show high bandwidth stability and read/write performance close to the values reported by the manufacturer.

NEST, a future use case for neural science, shows very good scalability up to high numbers of threads. The performance decreases only when the SMT region of the KNL is reached. The reason for the difference in timing in comparison to the Intel Xeon processor is not entirely clear yet.

In a next step, the network fabric (Omnipath) could be tested and used in order to reach even bigger network sizes that are not computable on one node any more.

7 Acknowledgements

A research project is never the work of one person alone. First of all, I would like to thank Prof. Dr. Dirk Pleiter for the possibility to work at this project. The discussions with him gave me further insight into the topic. Also, I am grateful to Dr. Marcus Richter who was my second supervisor.

One person that cannot be left unmentioned is Ivo Kabadshow who organized the Guest Student Program. He provided invaluable information about almost anything concerning C++ or LaTeX (especially tikz).

Last but not least, I would like to thank Bastian Tweddell, the system administrator of JULIA, who solved most of my problems instantly and provided perfect support at nearly any time of day.

References

[1] H. Bos, A. Morrison, A. Peyser, J. Hahne, M. Helias, S. Kunkel, T. Ippen, J. M. Eppler, M. Schmidt, A. Seeholzer, M. Djurfeldt, S. Diaz, J. Morén, R. Deepu, T. Stocco, M. Deger, F. Michler, and H. E. Plesser. **NEST 2.10.0**. https://doi.org/10.5281/zenodo.44222, doi:10.5281/zenodo.44222.

[2] N. Brunel. **Dynamics of Sparsely Connected Networks of Excitatory and Inhibitory Spiking Neurons**. 8(3):183–208. http://dx.doi.org/10.1023/A:1008925309027, doi:10.1023/A:1008925309027.

[3] **Intel® SSD DC P3600 Series Specifications**. http://www.intel.com/content/www/us/en/solid-state-drives/ssd-dc-p3600-spec.html.

[4] *Intel Xeon Phi processor high performance programming.*

[5] **Largest neuronal network simulation achieved using k computer | RIKEN**. http://www.riken.jp/en/pr/press/2013/20130802_1/.

[6] S. Lührs, D. Rohe, A. Schnurpfeil, K. Thust, and W. Frings. **Flexible and Generic Workflow Management**. pages 431–438. http://www.medra.org/servlet/aliasResolver?alias=iospressISBN&isbn=978-1-61499-620-0&spage=431&doi=10.3233/978-1-61499-621-7-431, doi:10.3233/978-1-61499-621-7-431.

[7] **pmbw – Parallel Memory Bandwidth Benchmark / Measurement – panthema.net**. https://panthema.net/2013/pmbw/.

[8] J. Vanderkay. **ezFIO – powerful, simple NVMe SSD benchmark tool**. http://www.nvmexpress.org/ezfio-powerful-simple-nvme-ssd-benchmark-tool/.

Independent Component Analysis on PLI Brain Images

Optimization and Parallelization

Fabian Preiß
Fakultät Mathematik und Naturwissenschaften
Uni Wuppertal
Germany
f.preiss@uni-wuppertal.de

Abstract *The method 3D-Polarized Light Imaging (3D-PLI) is a promising tool in mapping the fiber tracts of the human brain on both, small and large scales. In the imaging process the signal is degraded by several sources of noise, which reduction is of importance for the quality of the 3D-reconstruction. A parallelized Independent Component Analysis (ICA) based method was successfully adapted to the JURECA supercomputer environment and extended to seperate components based on their distribution being sub- or supergaussian.*

1 Introduction/Motivation

In the challenge of understanding the human brain, there is a variety of methods available in order to analyze structures on different scales (Figure 1.1). While methods like electron microscopy allow to show the structure of single axons and methods based on fluorescence microscopy can be used to identify local connections, they lack the ability of tracking more distant fiber tracts. On the other side classical dissection methods can be used in order to analyze the brain as a whole, however providing no further insight on the connections of the fiber tracts. More recently diffusion MRI has prooven itself as a successfull tool in tracking nerve fibers of the human brain, while unfortunately being restricted to resolutions in the mm-scale. Here 3D-Polarized Light Imaging (PLI) has been introduced as a new approach [1], reaching voxel dimensions of $1.3 \times 1.3 \times 70\,\mu m$ [4] while still being applicable to map the whole brain. This method is exploiting the birefringent properties of the myelin sheath surrounding nerve fibers[2].

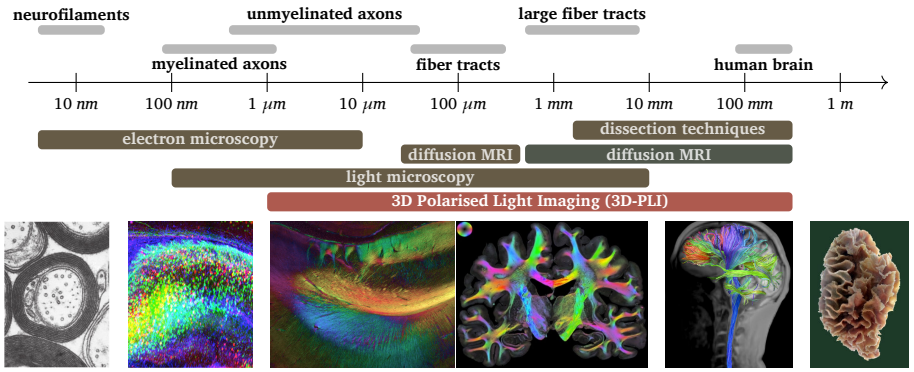


Figure 1.1: Available methods for the structure analysis of the human brain on different scales.

The imaging process for 3D-PLI takes place mainly on the Large Area Polarimeter (LAP) and the Polarizing Microscope. The increased resolution of the PLI-Microscope

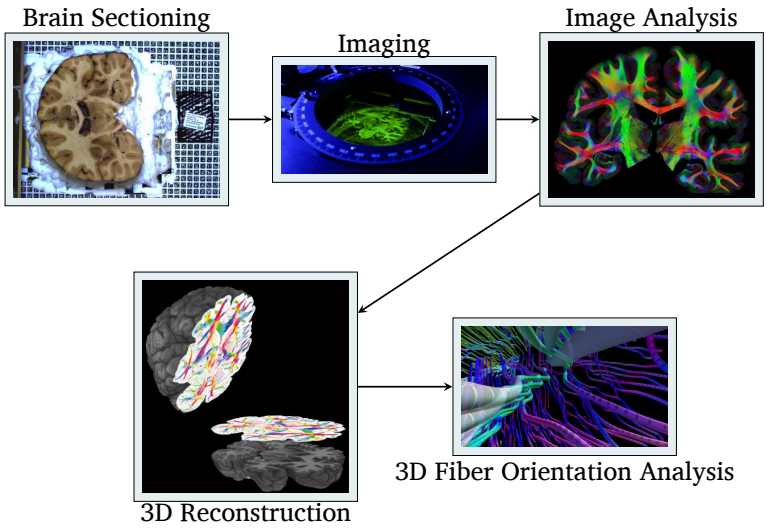


Figure 1.2: Workflow from sectioning the brain towards a 3D-Fiber map.

however adds additional challenges to the process of mapping a whole human brain, as the specimen has to be sectioned into ca. 3500 slices of about 70 µm while a typical slice takes up to 500 GB of space for the human brain[2]. In return similar efforts are taken in mapping the fiber tracts in rat and vervet monkey brains. The raw datasets are then processed in an image analysation pipeline to reduce unwanted effects, as well as extracting informations about the direction of the polarization, the transmittance of the probe and the retardation of the signal. Further steps combine the information gathered for every slice in order to achieve a 3D-reconstruction of the full brain and the 3D-structure of the nerve fiber connections (Figure 1.2).

This work is focused on applying an independent component analysis within the image analysation pipeline, which promises reduction of unwanted signals, as well as an improved feature extraction.

2 Problem Description

The deployed setup for the imaging process of 3D-PLI consists of a light source, 2 polarizers, a retarder, a stage for the probe and a camera device (Figure 2.1). The LAP furthermore allows a tilting of the probe, while the PLI-Microscope is driven by a motor for tile-wise scanning of the probe with overlapping fields of views[3]. The generated data is stored in HDF5 container files[13], typically in uncompressed 3d-arrays where 2 dimensions map to the xy-coordinates and the remaining dimension to the different angles under which the images were taken.

Due to the birefringence of the probe, the measured light intensity I varies in a sinusodial manner with a change of the polarizers rotation angle ρ. I is referred to as the light intensity profile

$$I = \frac{I_0}{2} \cdot [1 + \sin(2\rho - 2\varphi) \cdot \sin\delta]\,. \tag{2.1}$$

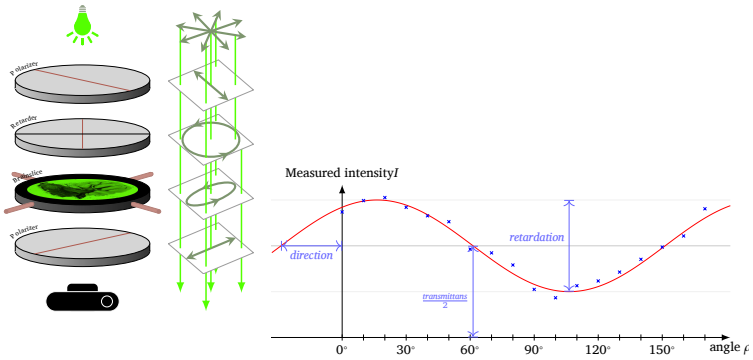


Figure 2.1: 3D-PLI setup for the Large Area Polarimeter (LAP)[3]. A narrow banded green LED light source with a wavelength of 525 ± 25 nm is used. The polarizers and retarder are rotated simultaneously by an automated system, furthermore the probe can be tilted by an angle of up to 8° in east, west, north and south direction.

The retardation δ is approximately given by

$$\delta \approx 2\pi \cdot \frac{d \cdot \Delta n}{\lambda} \cdot \cos^2 \alpha \tag{2.2}$$

where d is the thickness of the section, Δn the birefringence of the myelin, α the inclination angle of the fiber and λ the lights wavelength. On both PLI setups the images are typically taken under 18 different rotation angles from 0° to 170°.

However in the measurement process we naturally observe deviations from the previously described light intensity profile (eq. 2.1). These deviations can be traced back to an unknown mixing of different sources like camera noise, dust on the probe or the polarizers, as well as light scattering.

Here the independent component analysis (ICA) provides a statistical approach in order to estimate the different sources from the measured signal.

2.1 Independent component analysis (ICA)

The following section covers the theory behind the ICA method as well as some of the limitations that come with it.

2.1.1 The ideal cocktail party problem

Suppose we are in a room with n different people speaking simultaneously, then each of them will emmit a time dependent signal $s_j(t)$ where $j \in \mathbb{N}$ with $1 \leq j \leq n$. Now we place m microphones into this room, each of them recording a signal $x_i(t)$ with $i \in \mathbb{N}$ and $1 \leq i \leq m$. Ignoring time delays, a valid assumption seems to be, that the recorded signals can be expressed as a weighted sum of the emitted signals with weighting parameters a_{ij}[7]:

$$\begin{aligned} x_1(t) &= a_{11}s_1(t) + \ldots + a_{1n}s_n(t) \\ &\vdots \\ x_m(t) &= a_{m1}s_1(t) + \ldots + a_{mn}s_n(t) \end{aligned} \tag{2.3}$$

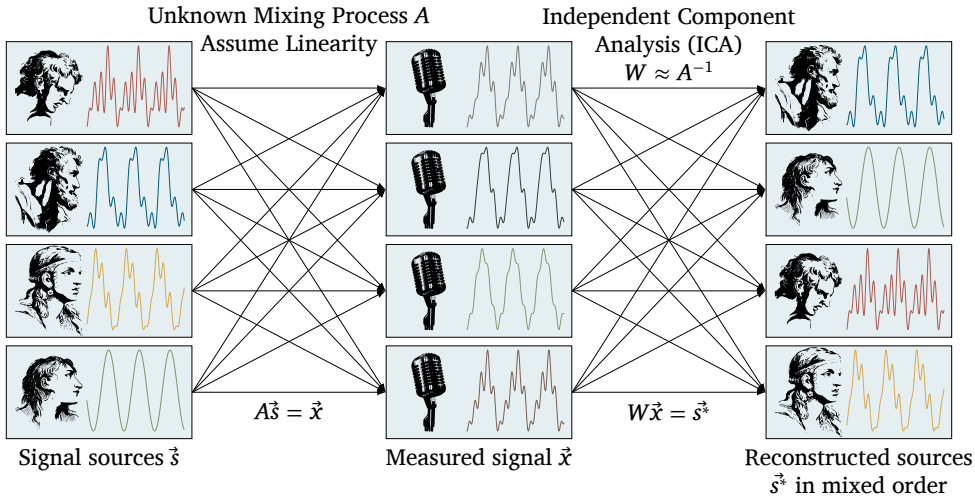


Figure 2.2: Schematic of the idealized cocktail party problem. Left: different people in a room speaking at the same time act as signal sources s_j. Center: A number of microphones spread in the room measure a linear mixture $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s}$ of these sources, where $\boldsymbol{A}$ is an unknown mixing matrix. Right: The source signals are reconstructed using an Independent Component Analysis (ICA) approach, where $\boldsymbol{W}$ is the estimated pseudoinverse of $\boldsymbol{A}$.

The problem of estimating the initial signals s_j while only having the recorded signals x_i available is famously known as the cocktail party problem. Using matrix notation and dropping the time index, equation 2.3 can simply be written as

$$\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s}\,. \tag{2.4}$$

If we were to know $\boldsymbol{A}$ and as long as the number of recordings is larger or equal to the number of initial signals ($n \leq m$) we could, for a full ranked matrix $\boldsymbol{A}$, solve equation 2.4 for $\boldsymbol{s}$ by the left-inverse $\boldsymbol{A}^{-1}_{\text{left}}$ of $\boldsymbol{A}$ with $\boldsymbol{s} = \boldsymbol{A}^{-1}_{\text{left}}\boldsymbol{x}$. Unfortunately the problem becomes considerably more difficult if we want to solve for $\boldsymbol{s}$ with unknown $\boldsymbol{A}$. The independent component analysis (ICA) approaches this problem by assuming that the initial signals s_j are statistically independent random variables (see 2.1.3), allowing for an estimation of $\boldsymbol{A}$ and in return for an approximation of its leftinverse, known as the unmixing matrix $\boldsymbol{W} \approx \boldsymbol{A}^{-1}_{\text{left}}$.

Furthermore it is helpful to assume that the observed variables x_i as well as the independent components $\boldsymbol{s}_j$ have zero-mean[7], if this is not fulfilled we can always center the individual components by substracting their mean $\hat{x_i} = x_i - \bar{x_i}$ and $\hat{s_j} = s_j - \bar{s_j}$.

2.1.2 Restrictions of the method

Before covering the theory behind the method the restrictions on the possible solutions should be kept in mind:

As both, the original signal $\boldsymbol{s}$ and the mixing matrix $\boldsymbol{A}$ are unknown, we can neither resolve the amplitude of the original signals, nor their order.

The first property can be understood by replacing one of the sources s_j with a multiple of itself $\lambda \cdot s_j$ where λ can be choosen with $\lambda \in \mathbb{C}\setminus\{0\}$. Dividing the corresponding

column with index j of the matrix a_{ij} by λ will yield the original vector $\boldsymbol{x}$ (see eq. 2.4). A natural way to resolve this is by fixing the magnitude of each source such that they have unit variance $\mathbb{E}\left\{s_j^2\right\} = 1$, this process is called whitening[7].

The second property can be shown by introducing a permutation matrix $\boldsymbol{P}_\pi$ and its inverse $\boldsymbol{P}_\pi^{-1}$ into equation 2.4 such that $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{P}_\pi^{-1}\boldsymbol{P}_\pi\boldsymbol{s}$. Now we have the new set of signals $\hat{\boldsymbol{s}} = \boldsymbol{P}_\pi\boldsymbol{s}$ that is simply the original vector $\boldsymbol{s}$ but in another order as well as the new mixing matrix $\hat{\boldsymbol{A}} = \boldsymbol{A}\boldsymbol{P}_\pi^{-1}$[7].

Another restriction has to be put on the sources themselves: As shown in the next section, we need the sources themselves to be distributed nongaussian.

2.1.3 Moments, correlation and dependence of random variables

As implied before, the ICA method exploits that the mixed signals $\boldsymbol{x}_i$ are no longer statistically independent. To clearify how this independency can be measured, the following section will give a short introduction to the necessary stochastics.

Expectation values and moments:

Given that a random variable X follows the probability density function $f_X(x)$, the expectation value $\mathbb{E}\{g\}$ of a function $g(x)$ that depends on this random variable is given by [11]

$$\mathbb{E}\{g(X)\} = \int_{-\infty}^{\infty} g(x) f_X(x)\, \mathrm{d}x. \tag{2.5}$$

In the discrete case, $P\{X = x_i\}$ denotes the probability of measuring the random variable X as x_i and equation 2.5 becomes

$$\mathbb{E}\{g(X)\} = \sum_i g(x_i) P\{X = x_i\}.$$

The expectation values $\mathbb{E}\{X^n\}$ in the continous case

$$m_n = \mathbb{E}\{X^n\} = \int_{-\infty}^{\infty} x^n f_X(x)\, \mathrm{d}x \tag{2.6}$$

and respectively in the discrete case

$$m_n = \mathbb{E}\{X^n\} = \sum_i x_i^n P\{X = x_i\} \tag{2.7}$$

are called the n-th moment of the random value X[11], where the first moment $\mu = \mathbb{E}\{X\}$ is also known as the mean. In return the central moments are given by

$$\mu_n = \mathbb{E}\left\{(X - \mu)^n\right\}. \tag{2.8}$$

The variance is defined by the second central moment $\sigma^2 = \mathbb{E}\left\{(X-\mu)^2\right\} = \mathbb{E}\{X^2\} - \mu^2$. It is useful to express higher moments in standart units with [8, p.99]

$$\alpha_n = \frac{\mu_n}{\sigma^n}.$$

The fourth central moment $\mu_4 = \mathbb{E}\left\{(X-\mu)^4\right\}$ is of particular interest for the independence analysis, as it defines the kurtosis. As the kurtosis is classically used as a

measurement of gaussianity and the fourth central moment of a normal distribution is $\mu_4 = 3\sigma^4$, we will use the more practical definition by Fisher[6, p.76] in standard units

$$\text{Kurt}[X] = \frac{\mu_4}{\sigma^4} - 3 \tag{2.9}$$

which is more accurately known as the excess kurtosis. With this definition a normal distribution the kurtosis $\text{Kurt}[X] = 0$, distributions with $\text{Kurt}[X] < 0$ are called platykurtic or sub-gaussian and have a relatively big part of their probability mass distributed in a *broad* center, whereas their tails are relatively flat. Vice versa distributions with $\text{Kurt}[X] > 0$ are called leptokurtic or super-gaussian and have comparably *fatter tails* than a normal distribution. For two independent random variables X and Y, the excess kurtosis furthermore satisfies the linear properties

$$\begin{aligned} \text{Kurt}[X+Y] &= \text{Kurt}[X] + \text{Kurt}[Y] \\ \text{Kurt}[\alpha X] &= \alpha^4 \text{Kurt}[X] \end{aligned} \tag{2.10}$$

with α being a scalar[7].

Independence and correlation:

Two given events $\mathcal{A}$ and $\mathcal{B}$ [11, p.27,32] are called independent, if their joint probability $P(\mathcal{A},\mathcal{B}) = P(\mathcal{A}|\mathcal{B})P(B) = P(\mathcal{B}|\mathcal{A})P(A)$ is given by the product of their individual probability:

$$P(\mathcal{A},\mathcal{B}) = P(\mathcal{A})P(\mathcal{B}) \tag{2.11}$$

Here $P(\mathcal{X})$ denotes the probability of an event $\mathcal{X}$ and $P(\mathcal{X}|\mathcal{Y})$ the conditional probability of an event $\mathcal{X}$, given that event $\mathcal{Y}$ occured. Thus equation 2.12 implies that the know

Corollary we find that two continous random variables X and Y are independent, if their joint probability density function (pdf) $f_{X,Y}(x,y)$ can be factorized into the according pdfs $f_X(x)$ and $f_Y(y)$[11, p.132]:

$$f_{X,Y}(x,y) = f_X(x) f_Y(y) \tag{2.12}$$

Furthermore, for a number of n random variables $X_1,\ldots,X_n$ the events are independent[11] if their joint pdf $f_{X_1,\ldots,X_n}(x_1,\ldots,x_n)$ fulfills eq. 2.13

$$f_{X_1,\ldots,X_n}(x_1,\ldots,x_n) = f_{X_1}(x_1)\cdot\ldots\cdot f_{X_n}(x_n)\,. \tag{2.13}$$

For two independent random variables X_1 and X_2 we find the relation

$$\mathbb{E}\{g_1(X_1)\cdot g_2(X_2)\} = \mathbb{E}\{g_1(X_1)\}\,\mathbb{E}\{g_2(X_2)\} \tag{2.14}$$

using equations 2.5 and 2.12[7].

Substituting $g_1(X_1) \to X_1$ and $g_2(X_2) \to X_2$ into equation 2.14 yields $\mathbb{E}\{X_1\cdot X_2\} - \mathbb{E}\{X_1\}\,\mathbb{E}\{X_2\} = 0$ and therefore demands the covariance of X_1 and X_2, defined as

$$\text{cov}(X_1,X_2) = \mathbb{E}\{X_1\cdot X_2\} - \mathbb{E}\{X_1\}\,\mathbb{E}\{X_2\} \tag{2.15}$$

to be zero. If $\operatorname{cov}(X_1, X_2) = 0$ the random variables X_1 and X_2 are called uncorrelated[11, p.153]. Similary n random variables $X_1, \ldots, X_n$ are called uncorrelated, if the covariance matrix, defined as

$$\Sigma(\boldsymbol{X}) = \left(\operatorname{cov}\left(X_i, X_j\right)\right)_{i,j=1,\ldots,n} = \mathbb{E}\left\{\boldsymbol{X}\boldsymbol{X}^T\right\} - \boldsymbol{\mu}\boldsymbol{\mu}^T$$

is the identity matrix I_n. However, equation 2.14 implies that uncorrelatedness is not a sufficient criterium for variables to be statistically independent. An improved estimate on the independence of random variables can be achieved, if higher moments are considered for equation 2.14. Here the ICA method heavily utilizes the fourth momentum known as kurtosis.

As mentioned before (2.1.2) the ICA method requires that at max. one of the sources s_j can be Gaussian distributed. The difficulties arising otherwise can be understood if we assume two underlying source signals s_1 and s_2 to be Gaussian distributed, with zero mean, uncorrelated thus $\Sigma(\boldsymbol{x}) = I_2$ and therefore $\boldsymbol{s} = \begin{pmatrix} s_1 & s_2 \end{pmatrix}^T \sim \mathcal{N}(0, I_2)$[10]. The linear mixing model leads to the observation of $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s}$, with the mixing matrix $\boldsymbol{A} \in \mathbb{R}^{2\times 2}$ and covariance matrix

$$\begin{aligned}
\Sigma(\boldsymbol{x}) &= \mathbb{E}\left\{\boldsymbol{x}\boldsymbol{x}^T\right\} - \mathbb{E}\left\{\boldsymbol{x}\right\}\mathbb{E}\left\{\boldsymbol{x}\right\}^T \\
&= \mathbb{E}\left\{\boldsymbol{A}\boldsymbol{s}\boldsymbol{s}^T\boldsymbol{A}^T\right\} - \mathbb{E}\left\{\boldsymbol{A}\boldsymbol{s}\right\}\mathbb{E}\left\{\boldsymbol{s}^T\boldsymbol{A}^T\right\} \\
&= \boldsymbol{A}\mathbb{E}\left\{\boldsymbol{s}\boldsymbol{s}^T\right\}\boldsymbol{A}^T - \boldsymbol{A}\mathbb{E}\left\{\boldsymbol{s}\right\}\mathbb{E}\left\{\boldsymbol{s}^T\right\}\boldsymbol{A}^T \\
&= \boldsymbol{A}\boldsymbol{A}^T.
\end{aligned}$$

If we introduce an arbitrary orthogonal (rotation/reflection) matrix $\boldsymbol{R} \in \mathbb{R}^{2\times 2}$ such that $\boldsymbol{R}\boldsymbol{R}^T = \boldsymbol{R}^T\boldsymbol{R} = I_2$, into our mixing model such that $\boldsymbol{x}' = \boldsymbol{A}\boldsymbol{R}\boldsymbol{s}$, we will again find $\boldsymbol{x}'$ distributed with zero mean and covariance matrix

$$\Sigma(\boldsymbol{x}') = \mathbb{E}\left\{\boldsymbol{x}' \cdot \boldsymbol{x}'^T\right\} = \mathbb{E}\left\{\boldsymbol{A}\boldsymbol{R}\boldsymbol{s}\boldsymbol{s}^T\boldsymbol{R}^T\boldsymbol{A}^T\right\} = \boldsymbol{A}\boldsymbol{R}\boldsymbol{R}^T\boldsymbol{A}^T = \boldsymbol{A}\boldsymbol{A}^T.$$

In consequence it for both $\boldsymbol{x}$ and $\boldsymbol{x}'$ our observation data would be distributed with $\mathcal{N}(0, \boldsymbol{A}\boldsymbol{A}^T)$, hiding whether the sources were mixed with $\boldsymbol{A}$ or $\boldsymbol{A}\boldsymbol{R}$.

An interpretation of this statement can be found using the central limit theorem, which states that the sum of n independent random variables s_j with $j \in 1, \ldots, n$ under certain conditions approaches a normal distribution as n increases[11, p.214]. Therefore the measured signals $\boldsymbol{x} = \boldsymbol{A}\boldsymbol{s}$ are distributed more gaussian, than the original sources s_j, giving us the possibility to search for a matrix $\boldsymbol{W} \approx \boldsymbol{A}_{\text{left}}^{-1}$ that minimizes gaussianity in $\hat{\boldsymbol{s}} = \boldsymbol{W}\boldsymbol{x}$. If the sources $\boldsymbol{s}$ however are already distributed gaussian, the actual solution $\boldsymbol{A}_{\text{left}}^{-1}$ can't be expected to decrease gaussianity in $\boldsymbol{A}_{\text{left}}^{-1}\boldsymbol{x}$, therefore rendering our approach useless for gaussian distributed sources.

2.1.4 Approaches in source estimation

In the section about the ideal cocktail party problem we stated, that the assumption of the source signals being statistically independent allows for an estimation of $\boldsymbol{W} \approx \boldsymbol{A}_{\text{left}}^{-1}$. A number of approaches in solving this problem can be motivated using information theory, the idea being that two statistically independent components $\begin{pmatrix} s_1 & s_2 \end{pmatrix}^T$ contain more information than their mixture $\begin{pmatrix} x_1 & x_2 \end{pmatrix}^T = \begin{pmatrix} a_{11}s_1 + a_{12}s_2 & a_{21}s_1 + a_{22}s_2 \end{pmatrix}^T$. This

is due to the fact, that some information between those two variables is now shared in their dependence. A measurement for this information is given by the Entropy H defined as

$$H(X) = -\int_{-\infty}^{\infty} f_X(x) \ln\left(f_X(x)\right) \, \mathrm{d}x \tag{2.16}$$

in the continous and

$$H(X) = -\sum_i P\{X = x_i\} \ln\left(P\{X = x_i\}\right) \tag{2.17}$$

in the discrete case[7]. In analogy to this, the joint entropy is given by

$$H(X,Y) = -\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} f_{X,Y}(x,y) \ln\left(f_{X,Y}(x,y)\right) \, \mathrm{d}x \, \mathrm{d}y \tag{2.18}$$

The method used in this implementation is based on maximizing the joint entropy (eq. 2.18) of $\hat{\boldsymbol{S}} = \Omega(\hat{\boldsymbol{s}})$ with $\hat{\boldsymbol{s}} = \boldsymbol{W}\boldsymbol{x}$ by adjusting the matrix $\boldsymbol{W} \approx \boldsymbol{A}_{\text{left}}^{-1}$, therefore leading to statistical independence between the source signals $\hat{s}_1, \ldots, \hat{s}_n$. Here Ω is introduced as a non-linear, bounded, continous and invertible cost function, it is typically choosen to be a sigmoidal function with $\Omega(\hat{\boldsymbol{s}}) = \frac{1}{1+e^{-\hat{s}}}$ or $\Omega(\hat{\boldsymbol{s}}) = \tanh(\hat{\boldsymbol{s}})$, flattening the distribution of $\hat{\boldsymbol{s}}$.

2.1.5 The infomax algorithm

An algorithm known to maximize the joint entropy of some estimated $\hat{\boldsymbol{S}}$ is known as the infomax algorithm. Before applying the infomax algorithm, the measured signals $\boldsymbol{x}$ are decorrelated with a variance of 1 as a preprocessing step, therefore reducing the search space. This is achieved in a process called whitening which is implemented by applying an eigenvalue decomposition to the covariance matrix of the signals $\boldsymbol{x}$[12]:

$$\Sigma(\boldsymbol{x}) = \mathbb{E}\{\mathrm{x} \cdot \mathrm{x}^T\} = \boldsymbol{Q} \cdot \Lambda \cdot \boldsymbol{Q}^T \tag{2.19}$$

Here $\boldsymbol{Q}$ is an orthogonal matrix with the eigenvectors and Λ a diagonal matrix containing the eigenvalues of $\Sigma(\boldsymbol{x})$. The according whitening matrix given by $\boldsymbol{D}_{\boldsymbol{x}} = \boldsymbol{E} \cdot \Lambda^{-1/2} \cdot \boldsymbol{E}^T$ can now be applied to $\boldsymbol{x}$ yielding the whitened measurements $\hat{\boldsymbol{x}} = \boldsymbol{D}_{\boldsymbol{x}} \cdot \boldsymbol{x} = \boldsymbol{D}_{\boldsymbol{x}} \cdot \boldsymbol{A} \cdot \boldsymbol{s}$ resulting in a new mixing matrix $\hat{\boldsymbol{A}} = \boldsymbol{D}_{\boldsymbol{x}} \cdot \boldsymbol{A}$ that is now orthogonal. In the following algorithm we assume that the measurements $\boldsymbol{x}$ have already been decorrelated.

The natural gradient version of the infomax algorithm iteratively adjusts the unmixing matrix $\boldsymbol{W} \approx \boldsymbol{A}_{\text{left}}^{-1}$ and the bias weight matrix $\boldsymbol{W}_0$ until $\hat{\boldsymbol{s}}$ reaches the intended precision. It is given by the scheme

$$\hat{\boldsymbol{s}} = \boldsymbol{W} \cdot \boldsymbol{x} + \boldsymbol{W}_0 \tag{2.20}$$

$$\Delta\boldsymbol{W} = \tau\left[\boldsymbol{I} + \left(1 - 2\hat{\boldsymbol{S}}\right) \cdot \hat{\boldsymbol{s}}^T\right] \cdot \boldsymbol{W} \tag{2.21}$$

$$\Delta\boldsymbol{W}_0 = \boldsymbol{I} - 2\hat{\boldsymbol{S}} \tag{2.22}$$

using the invertible cost function $\Omega(\hat{\boldsymbol{s}}) = \frac{1}{1+e^{-\hat{s}}}$ with $\hat{\boldsymbol{S}} = \Omega(\hat{\boldsymbol{s}})$, the learning rate τ and the identity matrix as the initial bias weight matrix $\boldsymbol{W}_0^{\text{initial}} = \boldsymbol{I}$[12]. To incooperate knowledge of the intensity distributions in gray and white matter, the cost function can furthermore be tuned with additional parameters b, ν and q such that $\Omega(\hat{\boldsymbol{s}}) = \frac{b}{\nu}\frac{1}{1+q \cdot e^{-b\hat{s}(k)}}$[12]. The learning rate is lowered with every iteration step.

Extended infomax

As illustrated in the work of Y. Wang [14], the infomax algorithm has some shortcomings when applied to 3D-PLI data, as it does not destinguish between super- and subgaussian distributed signals. An extension of the infomax algorithm that includes this distinction reads[9]

$$\Delta \boldsymbol{W} = \tau \left[\boldsymbol{I} - \boldsymbol{K} \cdot \hat{\boldsymbol{S}} \hat{\boldsymbol{s}}^T - \hat{\boldsymbol{s}} \hat{\boldsymbol{s}}^T \right] \cdot \boldsymbol{W} \tag{2.23}$$

where $\boldsymbol{K} \in \{-1, 0, 1\}^{m \times m}$ is a diagonal switching matrix for m measured signals with $\boldsymbol{K} = \text{diag}\,(k_1, \ldots, k_m)$ and

$$k_i = \begin{cases} -1 & \text{if signal } \hat{s}_i \text{ is subgaussian distributed} \\ 1 & \text{if signal } \hat{s}_i \text{ is supergaussian distributed} \end{cases}, \tag{2.24}$$

while the cost-function $\hat{\boldsymbol{S}} = \Omega\,(\hat{\boldsymbol{s}}) = \frac{b}{v} \tanh\,(\hat{\boldsymbol{s}} \cdot b)$ is used[5, 9]. The switching factors can be calculated by applying the signum function on the kurtosis.

2.2 ICA on PLI data

So far we only considered the independent component analysis for the classical cocktail party problem, where every measurement $x_i\,(t)$ is a time dependent and therefore one-dimensional signal. For 3D-PLI-data, the ICA algorithm is applied seperatedly on every brain slice, resulting in a 2-dimensional signal $x_\rho\,(u, v)$ for every angle ρ. As the implemented ICA algorithm does analyze the local surroundings of a pixel, the 2 spatial dimensions can be reduced by vectorization, thus $(x, y) \rightarrow (k)$, which allows us to drop the spatial coordinates from our notation altogether, yielding again the model $\boldsymbol{x} = \boldsymbol{As}$ we introduced in equation 2.4. Image slices, that are distributed over a set of tiles, like for the PLI-Microscope can be treated in the same way. In addition to this vectorization, the images are masked to allow a distinction between white and gray matter as well as the exclusion of image parts not including brain tissue.

As implied by equation 2.1, the intensity of the source signals varies in sinusoidal fashion with changing angle ρ. As described in the work of Tabbi[12], this allows for the application of certain constraints in the ICA algorithm, as only the signals of interest are expected to show this behaviour.

The goodness of fit for the function

$$f\,(\rho) = a_0 + a_1 \cdot \sin\,(2\varphi) + b_1 \cos\,(2\varphi) \tag{2.25}$$

to the coloumns of the estimated mixing matrix $\boldsymbol{W}^{-1}$ are then used to select the signals of interest from the reconstructed sources $\hat{\boldsymbol{s}}$.

As the image size for a single brain slice can exceed 500 GB [2], the dataset $\boldsymbol{x}$ is distributed over a number of processors using MPI within a python environment. Each processor executes one ICA iteration step on its share on $\boldsymbol{x}$, before permuting $\boldsymbol{x}$ as well as exchanging the learning rate, the weight matrix $\boldsymbol{W}$ and the bias weight matrix $\boldsymbol{W}_0$ cyclically with its neighbours. In the extended infomax implementation, informations about the kurtosis are furthermore exchanged.

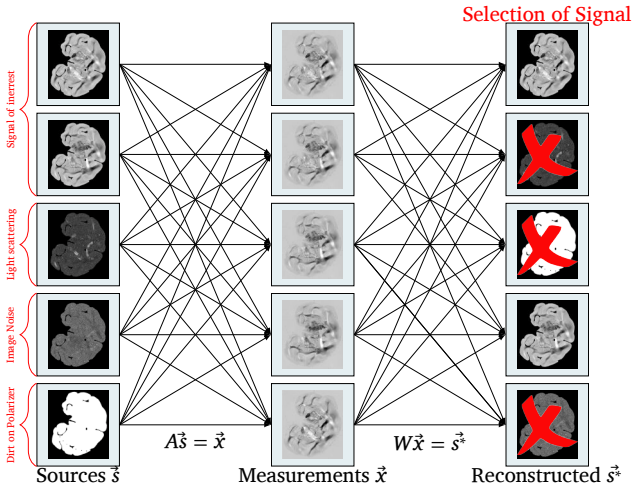


Figure 2.3: The optical measurements x_i of a 3D-PLI slice are modeled by a linear mixture $\boldsymbol{x} = \boldsymbol{As}$ of different unknown underlying physical sources s_j. After the application of the ICA method, a set of reconstructed sources $\hat{\boldsymbol{s}}$ is found allowing for a selection of signals of interest. The estimated mixing matrix $\boldsymbol{W}^{-1}$ can be applied on the selected signals, resulting in noise and artifact reduction.

3 Results

The parallel implementation of the infomax algorithm by Tabbi [12] has been extended to allow the optional use of the extended infomax algorithm as desribed in equation 2.23 and was adapted for the use on JURECA. In this process the initial program implementation has been significantly modified. The program has been tested on several comparably small datasets requiring between 576 MiB and a few Gigabyte of diskspace on up to 48 processors. For the small dataset the original implementation takes about 70 s on 48 processors, whereas the extended infomax implementation increases the runtime by about a factor of two. This does not come to a surprise, as especially the calculation of the Kurtosis is computationally expensive. It has to be noted, that the computation time for both methods heavily depends on the choice of a number of tuning parameters that depend on the measurement's setup as well as the observed and a more experienced choice might considerably change this statement.

Figure 3.1 shows, that the extended ICA implementation on JURECA successfully retrieved a number of independent source signals from the original measurements. The method impressively shows how otherwise hidden information that is distributed between the measurements can be retrieved and unveils significant structures that can be attributed to the birefringence of the myelin. In this example only the two first reconstructed sources certainly contain sources of interest, while the remaining ones contain mostly noise. However using similar tuning parameters on the larger dataset yielded one additional source that is clearly of interest and the order of the sources was not contained.

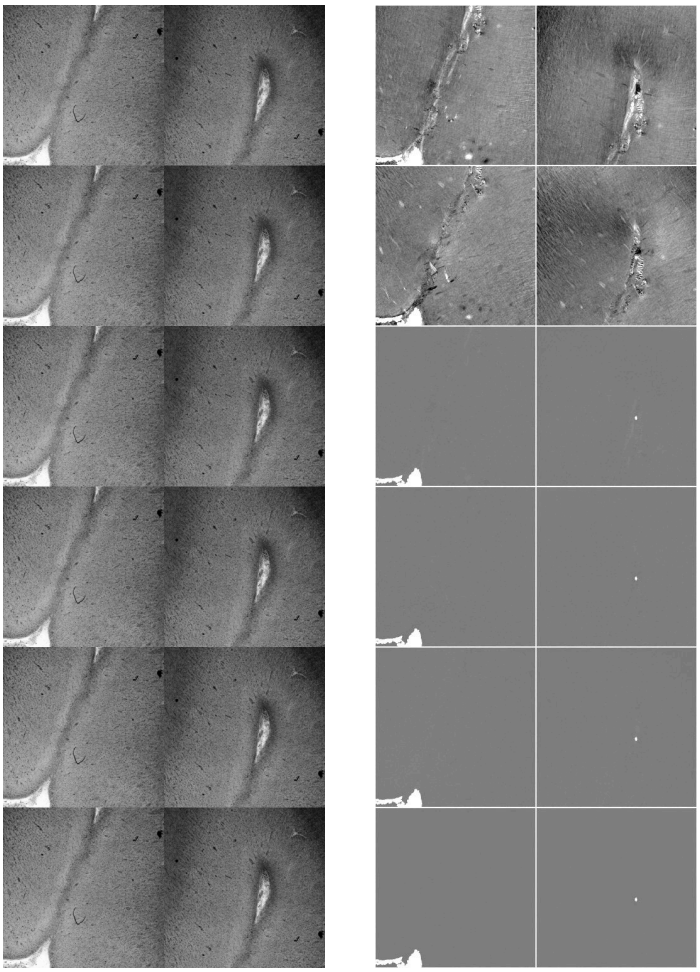


Figure 3.1: Extended ICA performed on a dataset of gray matter taken by the PLI-Microscope on two neighbouring tiles with a resolution of 2048 ×2048 px each. Both datasets, the measurements as well as the reconstruction take 576 MiB of discspace excluding the applied mask. Of the signals measured over 18 angles and the reconstructed 18 sources only 6 are shown and the contrast has been modified to highlight the areas of interest. Left: Measured signals shown with polarizer orientation ρ of $0°, 30°, \ldots, 150°$. Here the sinusoidal change of the measured light intensity I is only barely visible as the retardation is small compared to the overall change of the transmittance between the pixels. Right: 6 of the 18 reconstructed sources, where the first two images are components due to the image polarization, the remaining images (including the ones not displayed) to an overwhelming degree contain noise. It can be noted, that details not visible in the original set of images appear in the reconstructed sources, whereas some of the dark spots on the measured signals that can be attributed to dirt, are mostly gone in the reconstructed sources.

4 Conclusion

In this work the aim was to perform a statistical image analysis tool, specifically a signal-seperation on data generated by 3D-PLI systems. For the independent component analysis (ICA), an implementation based on the natural-gradient version of the Infomax algorithm was further modified to account for sub- and supergaussian distributed sources based on kurtosis estimation. While this extension comes with a considerably higher demand in computation, it could be shown that by extension of the underlying parallelization, this approach is still feasible for application on larger datasets.

In the future, computation time of larger datasets might be further reduced by estimating the unmixing matrix using only a certain number of samples from the original data. As the implementation depends on a significant amount of parameters and the variety of datasets this implementation was tested on was rather small, an improvement of the source seperation in both feature extraction and execution time can be expected by further studies of the parameter choices.

References

[1] M. Axer, K. Amunts, D. Grässel, C. Palm, J. Dammers, H. Axer, U. Pietrzyk, and K. Zilles. **A novel approach to the human connectome: Ultra-high resolution mapping of fiber tracts in the brain**. *NeuroImage*, 54(2):1091 – 1101, 2011. doi:10.1016/j.neuroimage.2010.08.075.

[2] K. Amunts, O. Bücker, and M. Axer. **Towards a multiscale, high-resolution model of the human brain**. In *Brain-Inspired Computing*, page 213 p. International Workshop, BrainComp 2013, Cetraro (Italy), 8 Jul 2014 – 11 Jul 2014, Springer, Jul 2014. doi:10.1007/978-3-319-12084-3_1.

[3] M. Axer, D. Graessel, M. Kleiner, J. Dammers, T. Dickscheid, J. Reckfort, T. Huetz, B. Eiben, U. Pietrzyk, K. Zilles, and K. Amunts. **High-Resolution Fiber Tract Reconstruction in the Human Brain by Means of Three-Dimensional Polarized Light Imaging**. *Frontiers in Neuroinformatics*, 5:34, 2011. doi:10.3389/fninf.2011.00034.

[4] M. Axer, S. Strohmer, D. Gräßel, O. Bücker, M. Dohmen, J. Reckfort, K. Zilles, and K. Amunts. **Estimating Fiber Orientation Distribution Functions in 3D-Polarized Light Imaging**. *Frontiers in Neuroanatomy*, 10:40, 2016. doi:10.3389/fnana.2016.00040.

[5] L. Breuer. *Identification of Neuromagnetic Responses for Real-Time Analysis in Magnetoencephalography*. PhD thesis, RWTH Aachen University, 2014.

[6] R. A. Fisher. *Statistical Methods for Research Workers*. Edinburgh, Scotland: Oliver & Boyd, 5 edition, 1934.

[7] A. Hyvärinen and E. Oja. **Independent component analysis: algorithms and applications**. *Neural Networks*, 13(4-5):411 – 430, 2000. doi:10.1016/S0893-6080(00)00026-5.

[8] J. F. Kenney and E. S. Keeping. *Mathematics of Statistics*, volume 1. D. Van Nostrand Company, 3 edition, 1954.

[9] T. W. Lee, M. Girolami, and T. J. Sejnowski. **Independent component analysis using an extended infomax algorithm for mixed subgaussian and supergaussian sources**. *Neural Comput*, 11(2):417–441, Feb 1999. https://www.ncbi.nlm.nih.gov/pubmed/9950738.

[10] A. Ng. **Independent Component Analysis**. Lecture notes, 2016.

[11] A. Papoulis. *Probability, Random Variables, and Stochastic Processes*. McGraw Hill, 1991.

[12] G. T. M. Tabbi. *Parallelization of a Data-Driven Independent Component Analysis to Analyze Large 3D-Polarized Light Imaging Data Sets*. PhD thesis, Bergische Universität Wuppertal, 2016. http://elpub.bib.uni-wuppertal.de/servlets/DocumentServlet?id=6010.

[13] The HDF Group. **Hierarchical Data Format, version 5**, 1997-2016.

[14] Y. Wang. **Comparison of Tensor Independent Component Analysis (ICA) with Joint ICA on Polarized Light Brain Images**. Master's thesis, Technische Universität Braunschweig, 2016.

Pixels, Matrices, and Circles on GPU

GPU Implementation of the Approach to Color Morphology Based on Loewner Order and Einstein Addition

Filip Srnec
Faculty of Science,
Department of Mathematics
University of Zagreb
Croatia
fsrnec.math@pmf.hr

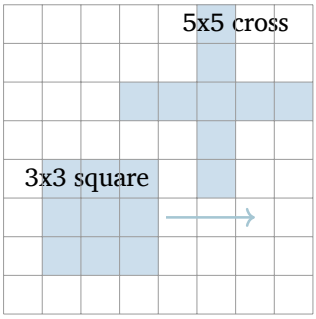
Abstract *In this paper, a GPU implementation of the approach to color morphology based on Loewner order and Einstein addition, firstly introduced by B. Burgeth and A. Kleefeld is presented. The implementation of basic gray-scale morphology operations erosion and dilation is demonstrated and used as a basis for the suggested approach. The conversion from RGB space to a symmetric matrix field is introduced and a fast way of comparing matrices using the Loewner ordering via solving the smallest enclosing circle of circles problem is demonstrated. Additionally, a way of implementing higher order morphological operations such as top-hats and gradients using the Einstein addition and two GPU devices is introduced.*

1 Introduction

Mathematical morphology is the theory of processing geometrical structures which was mainly introduced by Serra and Matheron ([15], [13]) in their work from the second half of the last century. Over the years, it has evolved into a powerful theory with applications in many areas of science, from digital image processing to medical imaging and geological science. Nowadays, mathematical morphology operations are fundamental techniques that are widely used in image processing.

The main idea of mathematical morphology is to probe the image with a structuring element, a geometrical shape that determines which pixels should be processed, to perform structural changes on pixel values. More precisely, a structuring element is used as a pixel mask and the morphological operation is performed only on pixels which are below the mask. The result of the operation in stored as a value of the pixel below the center of the structuring element. A few examples of possible structuring elements can be seen in Figure 1.

Mathematical morphology theory was originally developed for binary images, but later, theory for grey-scale images has also been established. Two fundamental morphological operations, which are formally defined for gray-scale images, are erosion and dilation. In lattice-theoretic framework with a total order, erosion and dilation rely on the notion of a minimum and a maximum. Intuitively, while calculating erosion, the goal is to find a minimal pixel value below the structuring element. On the other hand, while calculating dilation, the goal is to find a maximal pixel value below the structuring element. However, when it comes to color morphology, mathematical morphology theory for color images, defining basic morphological operations is not straightforward at all, due to the lack of a total order. There have been many attempts to establish the theory of morphological operations for color images, but none of them is unanimously accepted in the image processing community. For further information about different



(a) 3×3 square and 5×5 cross structuring elements.

0	1	0	127	30
1	1	1	58	80
0	1	0	30	8
127	127	127	80	127
25	58	126	8	126

(b) Probing the image with 3×3 cross structuring element.

Figure 1.1: Structuring elements in mathematical morphology.

approaches for defining color morphology framework, we refer the reader to [2], [18], and [21].

B. Burgeth and A. Kleefeld in [5] introduced a new approach to color morphology which is based on Loewner order among symmetric matrices and Einstein addition. It is shown that this approach has many advantages over previously discovered approaches. However, it is computationally expensive if implemented in a basic, straightforward way. In this paper, we present a GPU implmenetation of this approach. Including all the details about the introduced approach would be out of the scope of this paper, but all of them can be found in the original paper (see [5]). Having an implementation which can perform well in terms of time and memory usage, can be an useful foundation for future research. Also, it is time saving, since for real-world applications, time is crucial.

This paper is structured as follows: In Section 2 gray-scale morphology is introduced in a more formal way and a GPU implementation of basic morphological operations on gray-scale images is presented. The approach based on the work by B. Burgeth and A. Kleefeld and its GPU implementation is explained in detail in Section 4. Finally, in Section 5, higher order morphological operation are introduced, as well as their GPU implementations.

2 Gray-scale morphology

2.1 Definition

Gray-scale morphology is a term used for mathematical morphology theory applied to gray-scale images. Gray-scale images are images which consist of only one channel. More precisely, each pixel has a value between 0 and 255 which represents an intensity of the color *gray* related to that pixel. In gray-scale morphology, an image is defined as a function mapping the Euclidean space or an integer grid E into $\mathbb{R} \cup \{-\infty, \infty\}$. Structuring elements are also functions of the same format. As mentioned before, two fundamental operations of gray-scale morphology are erosion and dilation. Let f be a gray-scale image and b a structuring element. Dilation of f by b is given by

$(f \oplus b)(x) = \sup_{y \in E}[f(y) + b(x-y)]$. On the other hand, erosion is defined as $(f \ominus b)(x) = \inf_{y \in E}[f(y) + b(y-x)]$. For more about basic morphology operations, see [16] and [17]. Considering we have a total order on $\mathbb{R}$ and that pixels of gray-scale images we are processing have values in interval $[0, 255]$ associated with, we can refer to those morphological operations as finding a maximum pixel value below structuring element (dilation) and a minimum pixel value below structuring element (erosion).

2.2 GPU IMPLEMENTATION

When thinking about the implementation of basic gray-scale morphology operations one has to consider that all pixels of an image have to be processed which gives us a natural algorithm with complexity $\mathcal{O}(n^2)$ when processing the image with size $n \times n$. The idea is to run through every pixel and calculate the maximum (or the minimum) of pixels below the structuring element and store the result. Moreover, one can think of straightforward GPU implementation of the introduced algorithm using a natural 2-dimensional grid configuration to associate every pixel with one GPU thread. For example, in Figure 2.1 we show one possible grid configuration for performing morphological operation on a standard image processing testing image *Lena*. In this example, we assume that *Lena* consists of 8×8 (gray-scale) pixels and that we are using CUDA blocks of size 4×4 which is not the case in the actual implementation where we use 16×16 CUDA blocks.

If we look carefully at the introduced grid configuration and try to probe the image with an arbitrary structuring element, we would see that several problems can occur. Firstly, when performing structural changes on pixels that are on the edge of the image, the structuring element can go out of the borders of the image. Furthermore, some threads may use parts of the image that belong to "other" blocks. Problems that can occur are shown in Figure 2.2. While solving these problems one should think of GPU memory layout and good practices of writing CUDA code. We want to have proper memory access and also avoid branching so that each thread in one CUDA warp executes the same code.

One way of solving these problems is to calculate the index of a pixel which is currently processed by a thread and then use only pixels that are needed for calculating a structural change for a given pixel. This path obviously leads to branching which is exactly the thing we want to avoid. That is why we suggest the following: additional padding to the image needs to be provided. The idea is to expand the image with additional pixels on every side. The size of the padding should be equal to the $\lfloor dim/2 \rfloor$ where *dim* is the size of the side of the structuring element. One should consider that the side of the structuring element is usually an odd integer between 3 and 21 so the amount of additional memory we use is acceptable.

However, pixels that lie on the padding should not affect the result of the morphological operation. In order to assure that the result stays unchanged, the content of the padding should be determined on runtime. If dilation is performed (the maximum needs to be found), the padding should contain the minimal pixel value, 0, which corresponds to the color *black*. On the other hand, if erosion is performed (the minimum needs to be found), the padding should contain the maximal pixel value, 1, (we assume that pixel values are normalized to a value that belongs to the interval $[0, 1]$). Figure 2.3 shows the image after adding the additional padding. However, only pixels of the original image are processed.

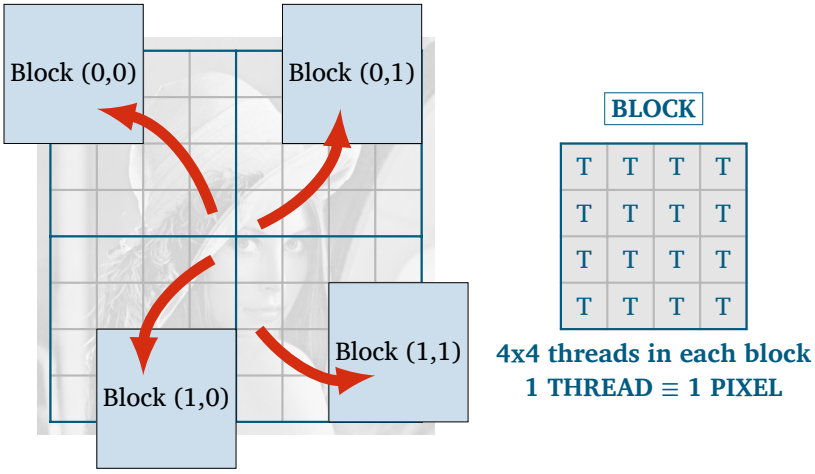


Figure 2.1: Natural 2×2 grid configuration for implementing mathematical morphology operations on GPU. For the state of the example, we assume that standard image processing *Lena* image has dimensions 8×8 and we use blocks that consist of 4×4 threads.

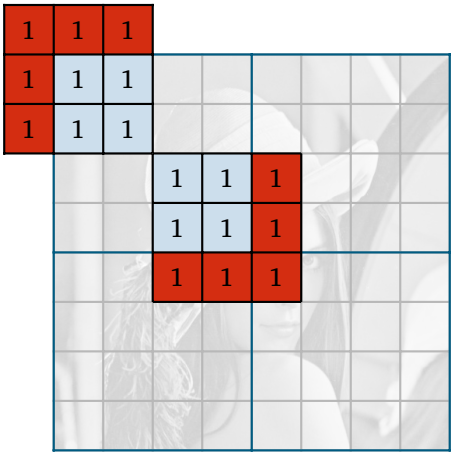


Figure 2.2: Problems that occur during implementation of the basic gray-scale morphology operations with 3×3 square structuring element after configuring the 2-dimensional grid. We can see that the structuring element can go out of borders of the image (for example, while calculating structural change of the pixel (0,0) of block (0,0)). Also, in some cases, for the calculation of a given morphological operation, we have to use parts of the image that we associate to "other" blocks which can lead to branching and unwanted memory access (for example, while calculating structural change of the pixel (3, 3) of block (0, 0)). Red color indicates states when problems occur.

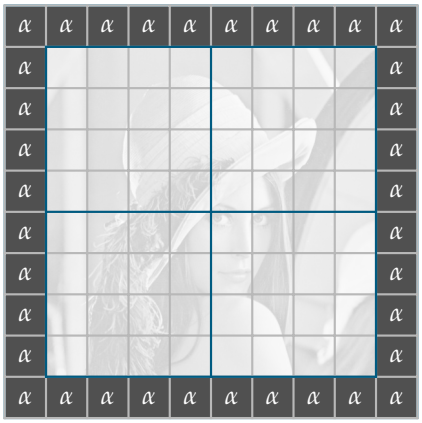
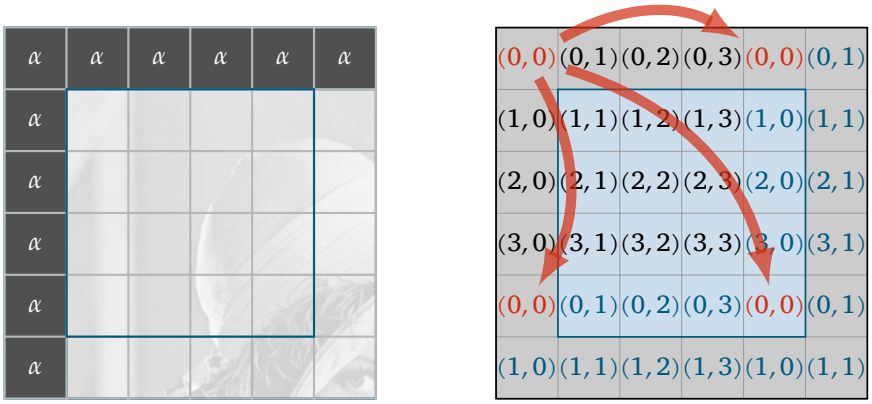


Figure 2.3: The image after adding the additional padding to assure that the structuring element will never go out of image borders. To assure that the result will stay unchanged, padding pixels should be filled with a maximal pixel value (1) for calculating erosion or a minimal pixel value (0) for calculating dilation.

To solve the second problem, optimizing memory access, one should consider using CUDA shared memory. Using shared memory in the right way usually leads to a performance boost. Shared memory is on-chip GPU memory, much faster than local and global memory. Shared memory latency is roughly 100× lower than uncached global memory latency. Moreover, it is allocated per thread block, so all threads in the block have access to the shared memory (see [9]). The main idea is to give a block the whole part of the image that it needs for computing morphological operations. That means that we have to store the part of the image associated with this block in the shared memory, as well as the padding for the structuring element. Therefore, the width of the tile stored in the shared memory will be $blockDim.x + padding$, and the height of the tile stored in shared memory will be $blockDim.y + padding$ where $padding = \lfloor dim/2 \rfloor$ and dim is a size of the side of the structuring element as before. Example of the shared memory tile associated to the block $(0, 0)$ from the *Lena* example is given in Figure 2.4a. To sum up, before a thread calculates a minimum or a maximum it loads a part of the image (with the padding) in the shared memory. Since we also have to consider the image padding, few of the threads have to store more data. We overcome this problem using the following code:

```
1 for (int i = threadIdx.y; i < sharedHeight; i += blockDim.y) {
2   for (int j = threadIdx.x; j < sharedWidth - 1; j += blockDim.x) {
3     smem[i * sharedWidth + j] = in_ptr[i * in_lda + j];
4   }
5 }
```

For further explanation, one can look at the Figure 2.4b which shows how the job of storing data in the shared memory is divided by GPU threads under assumption that we use a 3×3 structuring element and blocks of size 4×4 like in the previous *Lena* example. Also, one could notice that the interior *for* loop iterates while counter j is less than $sharedWidth - 1$. The reason is that the width of the shared memory tile is



(a) Tile in the shared memory associated to the block $(0,0)$ in our *Lena* example. The padding is also included in the tile, as well as parts of the image originally related to another block.

(b) Storing pixel values in the tile of shared memory related to each block in *Lena* example. Since 2-dimensional grid configuration is used, threads are labeled as ordered pairs (x,y) where x and y are appropriate block coordinates.

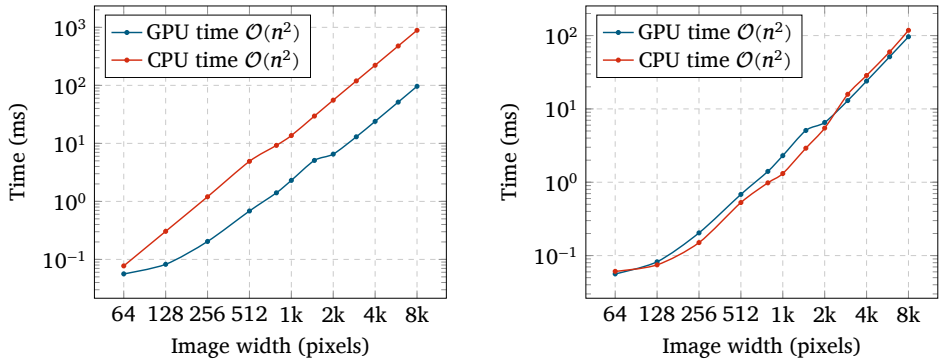
Figure 2.4: Using CUDA shared memory in the GPU implementation.

extended by 1 to prevent *bank conflicts*.

2.3 Results

In this section, we present performance measurement and results of our implementation of basic gray-scale morphology operations. For performance tests we used NVIDIA Tesla K40 GPU devices and Intel Xeon E5-2650 CPUs. We are comparing our approach with a straightforward CPU implementation which is based on the algorithm we mentioned in Section 2.2. We are measuring time needed for the gray-scale erosion using 3×3 square structuring element on the set of 12 gray-scale testing images with respective widths between values 64 and 8192. As we can see in Figure 2.5a, the GPU implementation outperforms straightforward CPU implementation which uses only 1 CPU thread. Moreover, times related to the CPU implementation are approximately 6 to 8 times longer compared to the time used by the GPU implementation, which is something that one could expect. Furthermore, since the log-log plot is used, one can make sure that the assumption of complexity $\mathcal{O}(n^2)$ was correct. On the other hand, Figure 2.5b shows that CPU implementation that uses 32 threads gives us times that are very close to our approach, considering we are using only 3×3 structuring element. This is expected if we think of the number of comparison operations on such a small structuring elements. However, when we compare the time needed for erosion with 7×7 structuring element on the same set of images (results are showed in Figure 2.6a) we can see that the GPU approach gives us a significant boost.

Last but not least, we compared our approach with two commonly used image processing libraries, *OpenCV* and *CImg*. We measured the time needed for erosion with



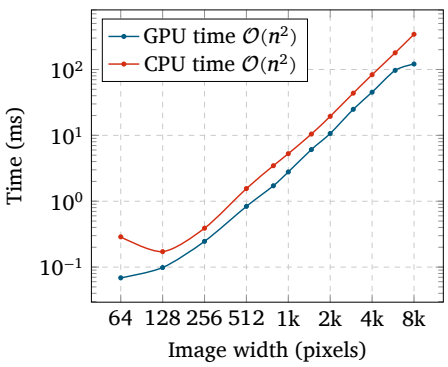
(a) Using 1 CPU thread

(b) Using 32 CPU threads

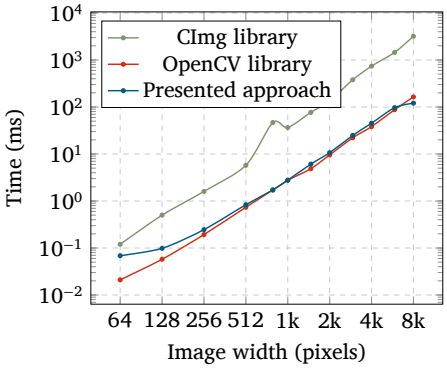
Figure 2.5: Gray-scale erosion using 3×3 square structuring element on GPU (NVIDIA Tesla K40) and CPU (Intel Xeon E5-2650).

7×7 square structuring element on the same set of images used in previous performance tests. As we can see on the Figure 2.6b, both our approach and *OpenCV* library erosion outperform the approach used in *CImg* library erosion which one could expect since *CImg* library does not use GPU for calculations. However, the time measured for our approach and for *OpenCV* library erosion are almost the same what we found very satisfactory.

To sum up, the introduced GPU implementation of basic morphological operations performs well even on the large images. For example, time needed for erosion with 3×3 structuring element on image with size 8192×8192 is still under 100 ms in our testing environment. In Figure 2.7 and Figure 2.8 we are showing results of morphological operations on the 512×512 *Lena* image.



(a) Using 32 CPU threads



(b) Using 3 different approaches: the GPU approach presented in this paper, CImg library erosion, OpenCV library erosion.

Figure 2.6: Gray-scale erosion using 7×7 square structuring element on GPU (NVIDIA Tesla K40) and CPU (Intel Xeon E5-2650).

(a) Erosion

(b) Dilation

Figure 2.7: Gray-scale morphological operations on 512×512 *Lena* testing image with 3×3 square structuring element.

(a) Erosion (b) Dilation

Figure 2.8: Gray-scale morphological operations on 512×512 *Lena* testing image with 7×7 diamond structuring element.

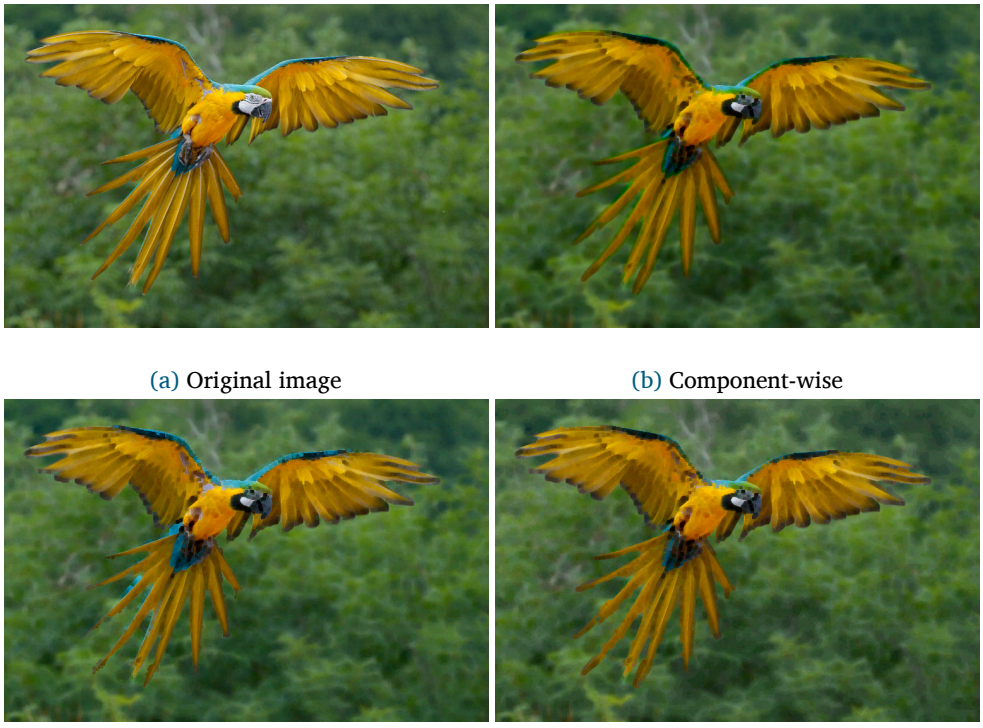
3 Color morphology

As we have already mentioned in the Section 1, finding the "right" approach to morphological operations for color (multi-channel) images is quite a challenging task. There is no unanimously acceptable way to define those operations. The question what is the supremum of two different colors is rather philosophical.

Over the years, many different approaches have been introduced. We will mention only a few of them. The basic approach is to perform the standard gray-scale morphology operation on each channel separately and then combine results in one final image. The second approach is to use standard lexicographic ordering between vectors containing color values. For example, if we have an image in *RGB* color space, each pixel is represented by a 3-dimensional vector containing values representing colors *red*, *green*, and *blue*, respectively. Standard lexicographic ordering will first compare *red* values then *green* and finally *blue* values. The third approach that we will mention is the approach which uses lexicographical cascades in *HSI* space. Since details about this approach are beyond the scope of this paper, for more information we refer the reader to [3].

In Figure 3.1 one can see and compare results of the performing erosion with 5×5 square structuring element based on each one of three introduced approaches on standard image processing testing image *Parrot*. One can notice how colors on the parrot's back and tail differ among approaches. Also, some differences can be seen on background colors. Deeper analysis of this approaches is out of the scope of this paper. In short, they are based on detecting "false" colors in result images, colors that do not appear in the original image.

Since both newly defined orders are total orders, implementation of the second and the third approach is similar to implementation of gray-scale morphology approach. The only difference is that the standard total order on $\mathbb{R}$ has to be replaced with the new order. The remaining GPU code stays the same. However, these approaches are not satisfactory in the sense that both of them require giving priority to color channels when defining lexicographic order. Our goal is to assure that all channels contribute equally in the calculation of morphological operation. This is exactly what we get using the approach of B. Burgeth and A. Kleefeld introduced in [5].



(a) Original image (b) Component-wise

(c) Lexicographic in *RGB* space (d) Lexicographic cascades in *HSI* space

Figure 3.1: Erosion with 5×5 square structuring element based on 3 different approaches on a standard image processing *Parrot* testing image.

4 Loewner morphology

In this section, we briefly describe the approach to color morphology based on Loewner order and Einstein addition introduced by B. Burgeth and A. Kleefeld in [5]. Later in the text, we will refer to this approach as *Loewner morphology*. The main idea is to code a color image as a symmetric matrix field and find a suitable order among matrices to calculate the infimum and the supremum in (gray-scale) definitions of dilation and erosion. From now on, we assume that we are processing color images with pixels represented by their *RGB* values (*red*, *green*, and *blue* values).

4.1 Conversion of an image to a matrix field

For the coding of a color image as a matrix field a variant of the *HSL*-color space (*hue*, *saturation*, and *luminance*) inspired by Ostwald's color solid form is used (see [14]). As presented in [5], as a first step, conversion from *RGB* to *m-HCL* color space needs to be done. This is accomplished by using a standard conversion from *RGB* to *HSL* color space which is well known in the image processing community (see [1]) and replacing saturation s with chroma c which is calculated by the following: $c = max\{r,g,b\} - min\{r,g,b\}$. After that, we have to modify luminance l by introducing $\tilde{l} = 2l - 1$. One can think of this conversion as a mapping from *RGB* unit cube, denoted by $\square$ (since we want to be consistent with the original paper) to $m-HCL$ bi-cone centered at 0 with a circular base of radius 1 with tips at $(0,0,1)$ and $(0,0,-1)$ (representing colors *white* and *black*, respectively) denoted by $\diamond$. In order to show this, we present the following mapping. Arbitrary point $p = (x,y,z) \in \diamond$ is obtained by $x = c\cos 2\pi h$, $y = c\sin 2\pi h$ and $z = \tilde{l}$.

The final step consists in applying the following mapping

$$\Psi : (x,y,z) \to A \tag{4.1}$$

where A is a symmetric matrix defined by

$$A = \begin{pmatrix} a & b \\ b & c \end{pmatrix} =: ([a,b],[b,c]) \in Sym(2) \tag{4.2}$$

with $a = \kappa(z-y)$, $b = \kappa x$ and $c = \kappa(z+y)$ where $\kappa = 1/\sqrt{2}$. One can show that Ψ is an isometry between the Euclidean space $\mathbb{R}^3$ and $Sym(2)$. Moreover, one-to-one mapping from the *RGB* space into the $\Psi(\diamond) = M \subset Sym(2)$ is established (for further explanation, see [5]).

In our implementation, both conversions (from image to matrix field and vice versa) are performed on the CPU since conversion operations require lots of condition checking what implies branching which is something we want to avoid while writing CUDA applications. However, converting pixel values can be independently done in parallel, so we use OpenMP to run the code in parallel.

4.2 Loewner order

As we mentioned before, a new order has to be introduced to find a minimal and a maximal matrix among a set of symmetric 2×2 matrices in order to perform the basic morphological operations erosion and dilation. For that purpose, we introduce the Loewner order. Let $A,B \in Sym(2)$ be symmetric 2×2 matrices. We say that $A \geq B$

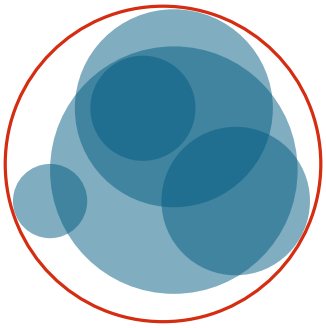


Figure 4.1: Smallest enclosing circle of circles problem. The goal is to find the smallest circle (a circle with the smallest radius) that contains all of a given set of circles (blue circles). The solution is marked with red color.

in terms of Loewner order if $A - B$ is positive semidefinite. One could show that matrices which are the result of mapping from the bi-cone $\diamond$ are in the Loewner interval $\kappa[-I, I] := \{A \in M \subset Sym(2) : -\kappa I \leq A \leq \kappa I\}$. The matrix $-\kappa I$ corresponds to the color *black* and the matrix κI corresponds to the color *white*.

When we think carefully about the Loewner order we can see that the Loewner order is not a total order since there exists a matrix which is not positive semidefinite. Therefore, the supremum and the infimum of the set of matrices is not necessary an element of the set since all the matrices in the set do not need to be even comparable. It is shown (see [6] and [4]) that the problem of finding the minimum and the maximum among a set of matrices, $\bar{A} := \max\{A_1, ..., A_n\}$, is equivalent to solving the smallest enclosing circle of circles problem on circles that are converted from the set of matrices. For the conversion, the following mapping is used: a circle which is a result of the conversion is a circle with center in (x, y) and radius r where $x = 2\kappa b$, $y = \kappa(c - a)$ and $r = \kappa(c + a)$. This mapping is obviously a bijective mapping.

4.3 Smallest enclosing circle of circles problem

Smallest enclosing circle of circles problem (from now on referred as *smallest circle problem*) is a problem of computing the smallest circle that contains all of a given set of circles in the Euclidean plane (see [7]). Figure 4.1 shows an example of the finding a solution to the Smallest circle problem. Since finding exact solution of this problem is computationally expensive, finding a "right" approach to use is an important task.

In our approach, we use a subgradient method to solve the problem numerically. It is a standard iterative optimization method usually used for solving convex minimization problems. Since the set of circles we are using for calculations is usually reasonably small (as mentioned before, the size of the side of the structuring element is usually between 3 and 21) there is no need to use more sophisticated methods. The idea is to formulate the smallest enclosing circle of circles problem as an unconstrained non-differentiable convex program:

$$\min f(x, y) = \max_{i=1,...,n} \sqrt{(a_i - x)^2 + (b_i - y)^2} + r_i$$

where $a_i = x_i + x$, $b_i = y_i + y$, (x_i, y_i) are the centers of the circles in the set and r_i radii of the circles in the set. For further explanation, see [24]. The algorithm itself is presented in Algorithm 1.

Algorithm 1 Subgradient method for finding the smallest enclosing circle

```
initialize starting point to (x_0 = 0, y_0 = 0);
α = 1.0;
k = 0;
while k < MAX_ITERATIONS do
  if |α| ≤ ALPHA_EPSILON then
    return circle with the center at (x_k, y_k) and radius f(x_k, y_k);
  end if
  compute subgradient (gradX, gradY) of f(x, y) at (x_k, y_k);
  use line search to update α;
  x_k = x_k + α · gradX;
  y_k = y_k + α · gradY;
  k = k + 1;
end while
```

However, this algorithm is not able to deal with negative radii. One can easily show that after performing a conversion from a matrix to a circle introduced in Section 4.2 some circles can end up with a negative radius since every symmetric matrix that participates in the conversion is a result of the mapping Ψ presented in Section 4.1 from the bi-cone $\diamond$ to $Sym(2)$. In order to solve this problem, we will not directly apply the introduced algorithm to matrices $A_1, ..., A_n$, but instead to their shifted versions $A_1 + \kappa I, ..., A_n + \kappa I$ (see [5] for further explanation). Furthermore, since basic morphological operations require calculation of both $\bar{A} := \max\{A_1, ..., A_n\}$ and $\underline{A} := \min\{A_1, ..., A_n\}$, it is worth mentioning that one can show that in terms of Loewner order the following equality is valid: $\underline{A} := \min\{A_1, ..., A_n\} = -\max\{-A_1, ..., -A_n\}$.

For the GPU implementation of the presented method we use the same grid configuration as for the gray-scale approach. The difference is that we do not store pixel values in GPU memory. In GPU memory we store matrices that were the result of the conversion presented in Section 4.1. Each GPU thread solves its own *smallest circle problem* locally using Algorithm 1, in the same way as it was calculating minimum and maximum operations in gray-scale morphology. To optimize memory access, we are again using CUDA shared memory, in the same way as we did in the gray-scale approach. However, there is one difference. This time, in shared memory we store circles (results of conversion from matrix) that are used for solving the *smallest circle problem*. Using this approach, we assure that all data needed for the subgradient method is stored in CUDA shared memory which gives us a major performance boost.

4.4 Pull-back mapping

Note that with the introduced *smallest circle problem* approach we are only calculating the pseudo supremum of the set of matrices. That means that the resulting maximum or minimum can be outside of the bi-cone $\diamond$, so the resulting matrix can not be converted back to the *RGB* pixel. However, it is proved (see [5]) that the solution of the *smallest*

circle problem is always in the unit ball with the center at $(0,0)$, denoted by $\mathcal{B}(0,1)$. So, the next step is finding the suitable mapping from the $\mathcal{B}(0,1)$ to the bi-cone $\diamond$. This mapping should be applied each time after the calculation of maximum and minimum in the terms of Loewner order. While constructing the mapping one should be aware of the fact that colors should not be too much affected by this mapping. That means that after applying the mapping, we should not get "false" colors.

For implementation of the pull-back mapping, we are using the theorem from the original work of B. Burgeth and A. Kleefeld (for detailed explanation and proof see [5] and [23]).

Theorem 1 *Let* $\hat{\lambda} \in \left(1, \sqrt{2}\right]$. *The mapping* Θ *given by*

$$\begin{aligned} \Theta \ :&\mathcal{B}(0,1) \to \mathbb{R}^3 \ , \\ &(x,y,z) \mapsto \left(\frac{1}{\mu}x, \frac{1}{\mu}y, \frac{1}{\mu}z\right) \ , \end{aligned}$$

with the unique real μ *satisfying the polynomial of degree* 11

$$\mu^{11} - \mu^{10} + 1 - \hat{\lambda} = 0 \ ,$$

where Θ *maps* $\mathcal{B}(0,1)$ *bijectively to the bi-cone* $\diamond$.

In the original paper, the constant $\hat{\lambda}$ is calculated using the following mapping from the unit ball $\mathcal{B}(0,1)$ to the bi-cone $\diamond$:

$$\hat{\lambda} = \hat{\lambda}(x,y,z) = 1 + \left(\sqrt{x^2+y^2} + |z|\right)^n \left(\frac{1}{\lambda} - 1\right) \ ,$$

where $n = 10$ and λ is defined by a mapping from unit ball to the bi-cone $\diamond$

$$\lambda = \lambda(x,y,z) = \begin{cases} 1 & \text{if } x,y = 0 \ , \\ \frac{\sqrt{(1+\Lambda^2)}}{1+\Lambda} & \text{otherwise} \end{cases}$$

with $\Lambda := \frac{|z|}{\sqrt{x^2+y^2}}$.

If we look carefully at bijective mappings used in point to matrix (Section 4.1) and matrix to circle (Section 4.2) conversions we can notice that an arbitrary point (x,y,z) from the Euclidean space is mapped to a circle with the center in (x,y) and radius $r = z$. That means no explicit conversion from a circle to a point in the unit ball $\mathcal{B}(0,1)$ is needed since the coordinates of a needed point have already been calculated in matrix to circle conversion. Therefore, the mapping introduced in Theorem 1 can be directly applied to the solution of the *smallest circle problem* stored as a circle. In our implementation every GPU thread already computes a solution of its local *smallest circle problem*, so before storing the result in matrix form, the introduced mapping should be applied on a point (x,y,r) where (x,y) is the center of the resulting circle and r its radius.

To apply the mapping, a unique real root of a polynomial $p(\mu) = \mu^{11} - \mu^{10} + 1 - \hat{\lambda} = 0$ should be found. In our implementation, we used Newton's iterative method. One can

assure that all the requirements for the convergence of Newtons's method are satisfied since all polynomials are in the class $C^{\infty}(\mathbb{R})$ and it is proved (see [5] and [23]) that p has unique real root.

The algorithm for mapping a point from the unit ball $\mathcal{B}(0,1)$ to a point on the bi-cone is presented in Algorithm 2.

Algorithm 2 Mapping a point from the unit ball $\mathcal{B}(0,1)$ to a point on the bi-cone $\diamond$

$\lambda = \lambda(x,y,z)$;
$\hat{\lambda} = \hat{\lambda}(x,y,z)$;
$p(\mu) = \mu^{11} - \mu^{10} + 1 - \hat{\lambda} = 0$;
$p'(\mu) = 11\mu^{10} - 10\mu^{9}$;
$k = 0$;
$\mu = 1.0$;
while $k <$ *MAX_ITERATIONS* **do**
  if $|p(\mu)| \leq$ *NEWTON_EPSILON* **then**
    return $\left(\frac{1}{\mu}x, \frac{1}{\mu}y, \frac{1}{\mu}z\right)$;
  end if
  $\mu = \mu - \frac{p(\mu)}{p'(\mu)}$
  $k = k + 1$;
end while

4.5 Implementation – all together

In this section, we will go through the presented GPU implementation of the basic morphological operations dilation and erosion based on the Loewner order once again to emphasize key-points and give the reader a better insight of our approach. A schematic view of the implementation can be seen in Figure 4.2.

The first step is to convert the input image that consists of *RGB* pixel values to a symmetric 2×2 matrix field which is performed on the CPU in parallel using OpenMP. After that, in order to optimize the GPU code, additional padding is added to the image, in a same way as described in Section 2.2 while describing GPU implementation of basic gray-scale morphology operations. In the other words, the amount of memory which is equal to the size of the image in the new matrix format with additional padding should be allocated. The next step is to fill the allocated memory with a maximum (if erosion is performed) or a minimum (if dilation is performed) value in terms of Loewner morphology. Those are matrices which represent color *white* (κI) and *black* ($-\kappa I$), respectively. Also, the converted image is copied to the GPU global memory.

After that, the whole structuring element is stored in the CUDA constant memory, special GPU memory which is provided by NVIDIA hardware. The whole constant memory space is cached. As a result, a read from constant memory cost one memory read from device memory only on a cache miss. Moreover, a request from constant memory is handled differently that other memory requests, due to the fact that while reading from constant memory CUDA uses different access patterns (see [8] for more). However, constant memory is only used for read-only data that will not change over the kernel execution. Since in this approach we assume that the structuring element stays the same during the calculation, this is exactly what we need.

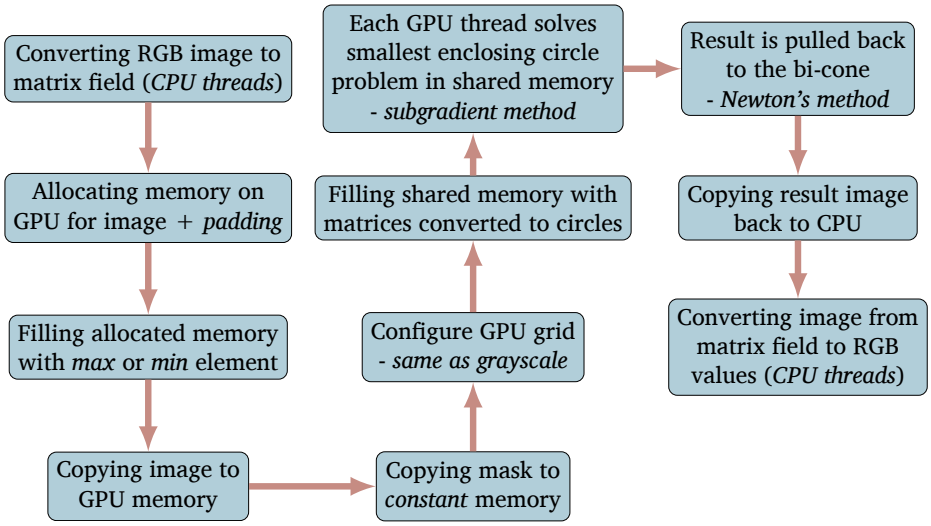


Figure 4.2: Schematic view of the implementation phases of the presented GPU implementation of the basic morphological operations based on Loewner order.

In the next phase, we need to configure a grid used by a kernel which performs the morphological operation. We use the same grid configuration that we presented in Section 2.2, while describing the implementation of gray-scale approach. Also, as explained in the last paragraph of Section 4.3, we are using CUDA shared memory in a slightly different way than in the gray-scale approach since we store circles which have already been prepared for solving the *smallest circle problem*. After that, each GPU thread solves the *smallest circle problem* independently with the explained subgradient method, using only data from the shared memory related to the block where the thread belongs to. As described in the Section 4.4, the mapping from the unit ball to the bi-cone needs to be applied.

Finally, we need to copy the resulting image back to the CPU and perform conversion from the symmetric matrix field back to the *RGB* format.

5 Higher order morphological operations

In our implementation, we included some well-known higher order morphological operations which will be introduced in this section.

5.1 Einstein addition

For the design of morphological operations such as top-hats and gradients, we need to define adequate addition and subtraction. However, one can easily show that the considered bi-cone $\diamond$ is not closed under standard matrix addition (for example, one can add the matrix representing the color *white* to itself).

In order to overcome those difficulties, B. Burgeth and A. Kleefeld, in [5], used a symmetric variant of the Einstein addition in Hilbert space. First, elements of the

bi-cone $\diamond$ are mapped to the unit ball $B(0,1)$. After that, the symmetric 2×2 matrix representation of the points on the unit ball is used (the mapping introduced in Section 4.1 is used). We denote this representation as $\Psi(\mathcal{B}(0,1))$. Let $A, B \in Sym(2)$ be symmetric 2×2 matrices and $\alpha_A = \sqrt{1 - ||A||^2}$. The Einstein addition is defined follows:

$$A \oplus_E B := \frac{1}{1 + \text{tr}(A \cdot B)} \left(A + \alpha_A B + \frac{\text{tr}(A \cdot B)}{1 + \alpha_A} A \right) .$$

However, it can be shown that the Einstein addition is not commutative. A commutative version is the Einstein co-addition which is given as

$$A \boxplus B := \frac{\alpha_A B + \alpha_B A}{\alpha_A + \alpha_B} \oplus_E \frac{\alpha_A B + \alpha_B A}{\alpha_A + \alpha_B} .$$

The subtraction is given as

$$A \boxminus B := A \boxplus (-B) = -B \boxplus A .$$

It can be shown that $\Psi(\mathcal{B}(0,1))$ is closed with respect to $\oplus_E$, $\boxplus$ and $\boxminus$. For details, see [19] and [20]. It is worth mentioning that result of the introduced addition should be mapped back to the bi-cone $\diamond$ using the introduced pull-back mapping from Section 4.4.

5.2 Definitions

In this section, definitions of higher order morphological operations, which were implemented in our approach, are presented. Let us assume that f is an image and b an arbitrary structuring element. Let us denote dilation with $f \oplus B$ and erosion with $f \ominus B$. We define the following operations:

- *opening*, $f \circ B = (f \ominus B) \oplus B$
- *closing*, $f \bullet B = (f \oplus B) \ominus B$
- *white top-hat*, $f \boxminus (f \circ B)$
- *black top-hat*, $(f \bullet B) \boxminus f$
- *self-dual top-hat*, $(f \bullet B) \boxminus (f \circ B)$
- *Beucher gradient*, $(f \oplus B) \boxminus (f \ominus B)$
- *internal gradient*, $\rho_B^- := f \boxminus (f \ominus B)$
- *external gradient*, $\rho_B^+ := (f \oplus B) \boxminus f$
- *morphological Laplacian*, $\Delta_b f := \rho_B^+ \boxminus \rho_B^-$
- *shockfilter*, $(trace(\Delta_b f) > 0)\ ?\ f \ominus B : f \oplus B$

5.3 Implementation

In this section, the GPU implementation of higher order morphological operations will be presented. As we can see in Section 5.2, higher order morphological operations are defined as compositions of basic morphological operations. Therefore, in the implementation we use the same GPU kernel for erosion and dilation presented before (refer to Section 4.5).

The implementation of opening and closing is straightforward since both erosion and dilation should be performed sequentially to get the right result.

On the other hand, for top-hats and gradients the situation is a bit different since they are defined as a subtraction (in terms of the Einstein addition) of two other morphological operations. One can notice that, for example, while implementing the Beucher gradient which is defined as $(f \oplus B) \boxminus (f \ominus B)$, dilation and erosion operations can be done in parallel since they are completely independent. For that reason, in our implementation we included an additional GPU device. The idea is to use two GPU devices to calculate both minuend and subtrahend for the Einstein subtraction in parallel and then to calculate the Einstein subtraction on one GPU. Here, we use CUDA peer-to-peer communication methods to exchange the data between two GPUs. This technique obviously cannot be used if we have access only to one GPU. In that case, in terms of optimization, we take advantage of CUDA streams to overlap computation and communication (see [10]).

When implementing the Einstein subtraction, we introduce the standard 2-dimensional grid configuration to divide the image in 16×16 CUDA blocks. Each thread has to calculate the Einstein difference between two matrices associated to the pixel in the same position in both result images. However, since the calculation is done for each thread independently, we can overlap the calculation of the difference and copying the image back to the CPU memory using CUDA streams like before. In this concrete case, we are using 4 CUDA streams.

6 Summary and outlook

In this paper, we have presented a GPU implementation of the new approach for color morphology based on the work of B. Burgeth and A. Kleefeld (see [5]). In our implementation, we used C++ with CUDA. As a result, there is a standalone C++ (CUDA) library for color morphology based on Loewner order and Einstein addition that contains all previously defined morphological operations (see Section 5.2). It can utilize up to 2 GPU devices. Since this is the first attempt of implementing morphology operations based on the Loewner order in terms of high performance computing in parallel, there is no existing code to compare our results with. However, there exists a MATLAB implementation originally written by A. Kleefeld (see [5]) which is much slower in computation time (as expected since the two technologies are incomparable) and therefore, the comparison with the GPU implementation would be unfair.

Color morphology operations have a large spectrum of applications. Firstly, one can think of many image filters (like shockfilter implemented in our library) which are based on morphological operations. Moreover, morphological operations are the basis for technique colloquially called “comification” — creating cartoon-like images from standard photos. Furthermore, higher order morphological operations have their applications in edge detection and image denoising which are very useful in many areas of science as well as in everyday life. That leads us to a conclusion that our final goal

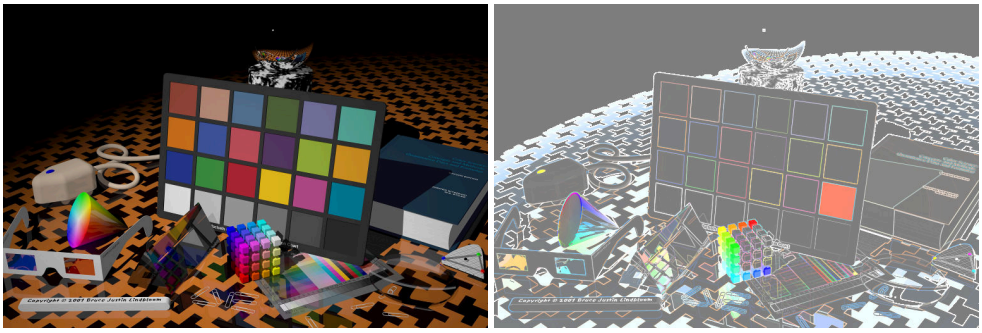
(a) Original image (b) result image

Figure 6.1: Internal gradient with diamond 7×7 structuring element on 512×512 *Pepper* image, 5 iterations.

should be finding a way to implement the presented technique for color morphology in a “right” way to get the most of the performance out of the technology that we have at our disposal. The reason is that, for real world applications, time matters.

The research of B. Burgeth and A. Kleefeld in terms of image processing using color morphology is still continuing and this will be the basis for future research. The main idea is to include additional filters in existing library code such as median filter, which is the basis of creating cartoon-like images mentioned before or amoeba quantile filters presented in a work of M. Welk, A. Kleefeld, and M. Breuß (see [23] and [22]). Also, there is a way to process multispectral images (images that consist of more than only 3 channels) via mathematical morphology presented by A. Kleefeld and B. Burgeth in [11]. However, one of the most challenging tasks is implementing mathematical morphology with adaptive structuring element. That means that, in this approach, the structuring element can change during the morphological operation. For more information about this approach, we refer the reader to the work by A. Kleefeld, M. Breuß, M. Welk, and B. Burgeth presented in [12].

In addition, we present only few examples of the morphological operations on standard image processing testing images generated by our library (see Figure 6.1, Figure 6.2, Figure 6.3, Figure 6.4, and Figure 6.5).



(a) Original image (b) Result image

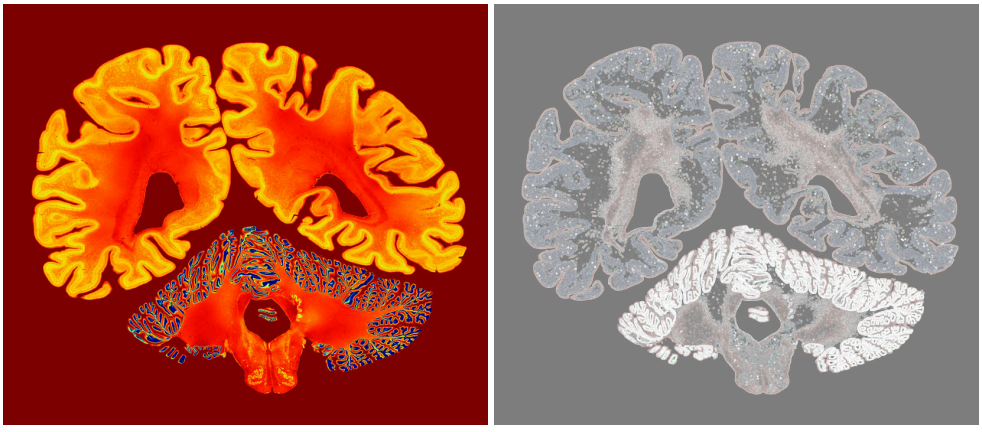
Figure 6.2: External gradient with square 3 × 3 structuring element on 768 × 512 *Color* image, 1 iteration.

(a) Original image (b) Result image

Figure 6.3: Dilation with diamond 21 × 21 structuring element on 768 × 512 *Parrot* image, 1 iteration.

(a) Original image (b) Result image

Figure 6.4: Shockfilter with square 3×3 structuring element on 512×512 *Lena* image, 20 iterations.



(a) Original image (b) Result image

Figure 6.5: Beucher gradient with square 5×5 structuring element on 6572×5711 *Brain* image, 2 iterations.

7 Acknowledgements

I sincerely thank my supervisor Dr. Andreas Kleefeld for all his generous support and help and for providing me a great insight into the area of image processing and mathematical morphology. Being a part of his project was a valuable and fulfilling experience. Also, I want to thank him for all the exciting table tennis matches we played during the time of this Guest Student Programme.

Furthermore, I am grateful to Dr. Ivo Kabadshow for the opportunity to be a part of the Guest Student Programme and for being available for all of my questions and concerns while I was working as a guest student in the JSC.

I am also thankful to Prof. Dr. Sc. Sanja Singer, Doc. Dr. Sc. Goranka Nogo, and Prof. Dr. Sc. Saša Singer for their enormous support and for directing me to this program.

Last but not least, I want to thank all of my colleagues for the amazing time we had during this program.

References

[1] M. K. Agoston. *Computer Graphics and Geometric Modeling: Implementation and Algorithms*. Springer, London, 2005.

[2] E. Aptoula and S. Lefèvre. **A comparative study on multivariate mathematical morphology**. *Pattern Recognition*, 40(11):2914–2929, 2007.

[3] V. Barnett. **The ordering of multivariate data**. *Journal of the Statistical Society*, A 139(3):318–355, 1976.

[4] B. Burgeth, A. Bruhn, N. Papenberg, M. Welk, and J. Weickert. **Mathematical morphology for tensor data induced by the Loewner ordering in higher dimensions**. *Signal Processing*, 87(2):277–290, 2007.

[5] B. Burgeth and A. Kleefeld. **An approach to color-morphology based on Einstein addition and Loewner order**. *Pattern Recognition Letters*, 47:29–39, 2014.

[6] B. Burgeth, N. Papenberg, A. Bruhn, M. Welk, C. Feddern, and J. Weickert. **Morphology for higher-dimensional tensor data via Loewner ordering**. In L. N. C. Ronse and E. Decencière, editors, *Mathematical Morphology: 40 Years On*, volume 30, pages 407–418. Springer, Dordrecht, 2005.

[7] K. Fischer and B. Gärtner. **The smallest enclosing ball of balls: combinatorial structure and algorithms**. *International Journal of Computational Geometry & Applications*, 14:341–378, 2004.

[8] N. Gupta. **Constant Memory in CUDA**, 2012. `http://cuda-programming.blogspot.de/2013/01/what-is-constant-memory-in-cuda.html`.

[9] M. Harris. **Using Shared Memory in CUDA C/C++**, 2013. `https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/`.

[10] M. Harris. **GPU Pro Tip: CUDA 7 Streams Simplify Concurrency**, 2015. `https://devblogs.nvidia.com/parallelforall/gpu-pro-tip-cuda-7-streams-simplify-concurrency/`.

[11] A. Kleefeld and B. Burgeth. **Processing multispectral images via mathematical morphology**. In *Visualization and Processing of Higher Order Descriptors for Multi-Valued Data*, pages 129–148. Springer, 2015.

[12] A. Kleefeld, M. Breuß, M. Welk, and B. Burgeth. **Adaptive filters for color images: median filtering and its extensions**. In *International Workshop on Computational Color Imaging*, pages 149–158. Springer, 2015.

[13] G. Matheron. *Eléments pour une théorie des milieux poreux*. Masson, Paris, 1967.

[14] W. Ostwald. *Die Farbenfibel*. Unesma, Leipzig, 1916.

[15] J. Serra. *Echantillonnage et estimation des phénomènes de transition minier*. PhD thesis, University of Nancy, France, 1967.

[16] J. Serra. *Image Analysis and Mathematical Morphology*, volume 1. Academic Press, London, 1982.

[17] J. Serra. *Image Analysis and Mathematical Morphology*, volume 2. Academic Press, London, 1988.

[18] J. Serra. **The false colour problem**. In *Mathematical Morphology and Its Application to Signal and Image Processing, Proceedings of the 9th International Symposium on Mathematical Morphology*, volume 5720, chapter 2, pages 13–23. Springer, Heidelberg, 2009.

[19] R. U. Sexl and H. K. Urbantke. *Relativity, Groups, Particles: Special Relativity and Relativistic Symmetry in Field and Particle Physics*. Springer, Wien, 2001.

[20] A. A. Ungar. *Einstein's Special Relativity: The Hyperbolic Geometric Viewpoint*. Conference on Mathematics, Physics and Philosophy on the Interpretations of Relativity, II. Budapest, 2009.

[21] J. J. van de Gronde and J. B. T. M. Roerdink. **Group-invariant frames for colour morphology**. In C. L. Luengo Hendriks, G. Borgefors, and R. Strand, editors, *Mathematical Morphology and Its Applications to Signal and Image Processing (Proceedings of the 11th International Symposium on Mathematical Morphology, Uppsala, Sweden, May 27–29)*, volume 7883, pages 267–278. Springer, Berlin, 2013.

[22] M. Welk, A. Kleefeld, and M. Breuß. **Non-adaptive and amoeba quantile filters for colour images**. In *International Symposium on Mathematical Morphology and Its Applications to Signal and Image Processing*, pages 398–409. Springer, 2015.

[23] M. Welk, A. Kleefeld, and M. Breuß. **Quantile filtering of colour images via symmetric matrices**. *Mathematical Morphology-Theory and Applications*, 1(1):136–174, 2016.

[24] S. Xu, R. M. Freund, and J. Sun. **Solution methodologies for the smallest enclosing circle problem**. *Computational Optimization and Applications*, 25:283–292, 2003.

Gray-Scott simulations with SDC and DUNE

Mia Jukić
Department of Mathematics
Faculty of Science,
University of Zagreb
Croatia
mia.jukic2@gmail.com

Abstract *Reaction and diffusion of chemical species can produce a variety of patterns. The Gray-Scott equation models such a reaction. In order to do numerical simulation of Gray-Scott model, for this work, Spectral Deferred Correction (SDC) is used for discretization of the time domain and then Finite Element Method (FEM) is used for solving system of PDEs.*

1 Introduction

A reaction-diffusion system is a mathematical model which is usually associated with change in space and time of the concentration of one or more chemical substances. This system is usually represented with systems of partial differential equations which consists of a reaction part that describes local chemical reactions and a diffusion part that causes the substances to spread out. One of the reasons why these reactions are of such an interest is that they can produce variety of patterns and behaviours, often similar to ones that can be found in nature.

In order to peform get numerical simulations of the Gray-Scott model, we use the spectral deferred correction (SDC) method for discretization of the time domain. One of the advantages of SDC is that the numerical method used to approximate the correction equations can be low-order accurate, while the solution after many iterations can in principal be of arbitrarily high-order of accuracy. This has been exploited to create SDC methods that allow the equations to be split into stiff and non-stiff parts that can be treated either implicitly or explicitly. In our work, we treat the reaction part explicitly and the diffusion part implicitly. That way we get a system of elliptic partial differential equations which we will solve with the Finite Element Method.

The outline of this paper is as follows. In Section 2 we give a short description of SDC, together with the semi-implicit version for a general class of ODEs. In Section 3 we give a short introduction to the Finite Element Method and then, in Section 4, we explain the basic idea behind dune-PFASST++, the software that we used. In Section 5 we give results of our implementation of a multi-component model and we also test convergence in space and time. Finally, in Section 6, we describe the Gray-Scott model and show the results of numerical simulations.

2 Spectral deferred corrections

Consider the ODE initial value problem (IVP)

$$\begin{aligned} \frac{\partial u}{\partial t} &= f(u(t)), \quad t \in [0, T], \\ u(0) &= u_0, \end{aligned} \tag{2.1}$$

where $t \in [0, T]$, $u_0, u(t) \in \mathbb{R}^N$ and $f : \mathbb{R}^N \times \mathbb{R} \to \mathbb{R}^N$. To describe *SDC*, it is convenient to use the equivalent Picard integral form of (2.1) which is given by

$$u(t) = u_0 + \int_0^t f(u(s))ds.$$

We now focus on a single timestep $[T_n, T_{n+1}]$, which is divided into subsets by defining a set of quadrature nodes on the interval. Here we consider Radau quadrature and denote $M+1$ nodes $\mathbf{t} := (t_m)_{m=0,\dots,M}$ such that $T_n < t_0 < t_1 \dots < t_M = T_{n+1}$. The SDC method begins by first computing a provisional solution $\mathbf{U}^0 = [U_1^0, \dots, U_M^0]$ at each of the intermediate nodes with $U_m^0 \approx u(t_m)$. In order to derive equations for the intermediate solutions U_j, we define quadrature weights

$$q_{m,j} := \int_{T_n}^{t_m} l_j(s)ds, \quad m = 0, \dots, M, \quad j = 0, \dots, M,$$

where $(l_j)_{j=0,\dots,M}$ are the Lagrange polynomials defined by the nodes $\mathbf{t}$. We obtain

$$U_m = U_m^0 + \sum_{j=0}^{M} q_{m,j} f(U_j), m = 0, \dots, M. \tag{2.2}$$

For a more compact notation, we define the *integration matrix* $\mathbf{Q}$ to be the $M+1 \times M+1$ matrix consisting of entries $q_{m,j}$. If we denote by

$$\mathbf{U} := [U_0, \dots, U_M]^T$$
$$\mathbf{F(U)} := [F_0, \dots, F_M]^T := [f(U_0), \dots, f(U_M)],$$

then we can write set of equations (2.2) more compactly as

$$\mathbf{U} = \mathbf{U}_0 + \mathbf{Q}\ \mathbf{F(U)}.$$

After applying Picard iterations in order to approximate the solution of the upper problem we get

$$\mathbf{U}^{k+1} = \mathbf{U}^k + \mathbf{U}_0 - (\mathbf{I}_N - \mathbf{QF})(\mathbf{U}^k).$$

After preconditioning using a simpler integration rule Q_Δ, we derive the next formula for solution in $k+1^{\text{th}}$ iteration:

$$\mathbf{U}^{k+1} = \mathbf{U}_0 + \mathbf{Q}_\Delta \mathbf{F}(\mathbf{U}^{k+1}) + (\mathbf{Q} - \mathbf{Q}_\Delta)\mathbf{F}(\mathbf{U}^k)$$

We define by

$$s_{m,j} := \int_{t_{m-1}}^{t_m} l_j(s)ds, \quad m = 1, \dots, M$$

the quadrature weights for node to node integration and as $\mathbf{S}$ the matrix consisting of entries $s_{m,j}$. It depends on the integration rule Q_Δ whether the iteration is explicit or implicit. For example, if we take Q_Δ which corresponds to the forward Euler method, as depicted in Figure 2.1a, we get

$$U_{m+1}^{k+1} = U_m^{k+1} + \Delta t_m \left[f(U_m^{k+1}) - f(U_m^k) \right] + S_m^k\ . \tag{2.3}$$

On the other hand, if we take Q_Δ which corresponds to the backward Euler method, as depicted in Figure 2.1b, we get

$$U_{m+1}^{k+1} = U_m^{k+1} + \Delta t_m \left[f(U_{m+1}^{k+1}) - f(U_{m+1}^k) \right] + S_m^k\ , \tag{2.4}$$

where in both equations S_m^k denotes the m^{th} row of $\mathbf{SF}(\mathbf{U}^k)$ and $\Delta t_m := t_{m+1} - t_m$. The process of solving (2.3) or (2.4) at each node is referred to as an *SDC sweep*. SDC methods for ODEs were first introduced in [1], and were subsequently refined and extended e.g. in [4].

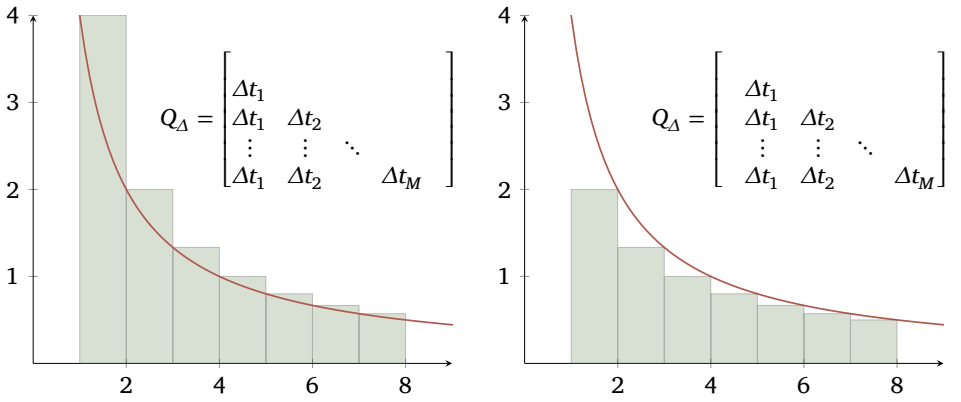


Figure 2.1: Left: Quadrature rule with explicit Q_Δ. Right: Quadrature rule with implicit Q_Δ

2.1 IMEX SDC SCHEME

In many practical applications, systems of ODEs that contains both stiff and non-stiff elements must be solved. Such an IVP is given below

$$\begin{aligned} \frac{\partial u}{\partial t} &= f^E(u(t)) + f^I(u(t)) \quad t \in [0,T] \\ u(0) &= u_0. \end{aligned} \tag{2.5}$$

Two straight-forward approaches naturally arise: to solve the whole system using an implicit scheme or to use an explicit method with a short time step. The basic premise of an IMEX solver is to provide a compromise – the stiff parts of the system are solved with an implicit solver to ensure stability, while the non-stiff parts are solved with an explicit solver to reduce computational load. SDC method is easily modified to create semi-implicit or IMEX schemes. Equations (2.3) and (2.4) can be easily modiefied to give a semi-implicit scheme:

$$U_{m+1}^{k+1} = U_m^{k+1} + \Delta t_m \left[f^E(U_m^{k+1}) - f^E(U_m^k) \right] + \Delta t_m \left[f^I(U_{m+1}^{k+1}) - f^I(U_{m+1}^k) \right] + S_m^k. \tag{2.6}$$

3 GENERALIZATION TO PDE

In this section we look at a general reaction-diffusion system which can be represented in the general form

$$\frac{\partial u}{\partial t} = D\Delta u + R(u),$$

where $u = u(x,t)$ represents the unknown function, D is a diagonal matrix of diffusion coefficients and R is a reaction term. We will treat the diffusion part implicitly and the reaction term explicitly. After applying IMEX SDC scheme we get following semi-discrete equation

$$\begin{aligned} U_{m+1}^{k+1}(x) - D\Delta t_m \Delta U_{m+1}^{k+1}(x) &= U_m^{k+1}(x) + \Delta t_m R(U_m^{k+1}(x)) \\ &\quad - R(U_m^k(x)) - D\Delta(U_{m+1}^k(x)) + S_m^k \\ &=: b_m^k(x), \quad m = 0, \dots, M-1. \end{aligned} \tag{3.1}$$

We can see that solution $\mathbf{U}^{k+1}$ at each node t_m is a solution of a system of elliptic partial differential equations. In order to describe a method for solving such equations we will focus on this elliptic PDE:

$$\begin{aligned} -D\Delta u(x) + u(x) &= f(x) \quad x \in \Omega \\ u(x) &= 0 \quad x \in \partial\Omega. \end{aligned} \tag{3.2}$$

A function $u \in C^2(\Omega) \cap C(\overline{\Omega})$ satisfying (3.2) would be called *classical solution* of this problem. The theory of partial differential equations tells us that (3.2) has a unique classical solution, provided that f and $\partial\Omega$ are sufficiently smooth. However, in many applications one has to consider equations where these smoothness requirements are violated, and for such problems the classical theory is inappropriate. To begin, let us suppose that u is a classical solution of (3.2). Then, if we multiply the equation with $v \in C_0^1(\overline{\Omega})$ and integrate over Ω we get

$$-D\int_\Omega \Delta u(x)v(x)dx + \int_\Omega u(x)v(x)dx = \int_\Omega f(x)v(x)dx.$$

Upon integration by parts in the first integral and noting that $v = 0$ on $\partial\Omega$, we obtain

$$D\int_\Omega \nabla u(x)\nabla v(x)dx + \int_\Omega u(x)v(x)dx = \int_\Omega f(x)v(x)dx.$$

In order for this equation to make sense, we do not need to satisfy that $u \in C^2(\Omega)$. It is sufficient that $u \in L^2(\Omega)$, $\nabla u \in L^2(\Omega)^d$ and $f \in L^2(\Omega)$. Having in mind that that $u = 0$ at boundary, it is natural to seek solution $u \in H_0^1(\Omega)$, where H_0^1 is defined as

$$H_0^1(\Omega) = \left\{u \in L^2(\Omega) : \nabla u \in L^2(\Omega)^d, u = 0 \text{ on } \partial\Omega\right\}.$$

This leads to following problem: find $u \in H_0^1(\Omega)$ such that

$$D\int_\Omega \nabla u(x)\nabla v(x)dx + \int_\Omega u(x)v(x)dx = \int_\Omega f(x)v(x)dx, \quad \forall v \in C_0^1(\Omega). \tag{3.3}$$

It can be shown that if we expand our space of test functions from $C_0^1(\Omega)$ to $H_0^1(\Omega)$ that there exists the unique solution of described problem and that solution we call weak solution of (3.2). Once we formulated our problem, the next question is how to find a weak solution. The first step is to replace $H_0^1(\Omega)$ by a finite-dimensional subspace $V_h \subset H_0^1(\Omega)$. Then we consider the following approximation of (3.3): find $u_h \in V_h$ such that

$$D\int_\Omega \nabla u_h(x)\nabla v(x)dx + \int_\Omega u_h(x)v_h(x)dx = \int_\Omega f(x)v_h(x)dx, \quad \forall v_h \in V_h. \tag{3.4}$$

As we said, we choose V_h to be finite-dimensional space so we can write

$$\dim V_h = N, \quad V_h = \left[\{\varphi_1, \dots, \varphi_N\}\right].$$

Expressing the approximate solution u_h as a linear combination of basis functions, we can write

$$u_h(x) = \sum_{j=1}^{N} U_j\varphi_j(x),$$

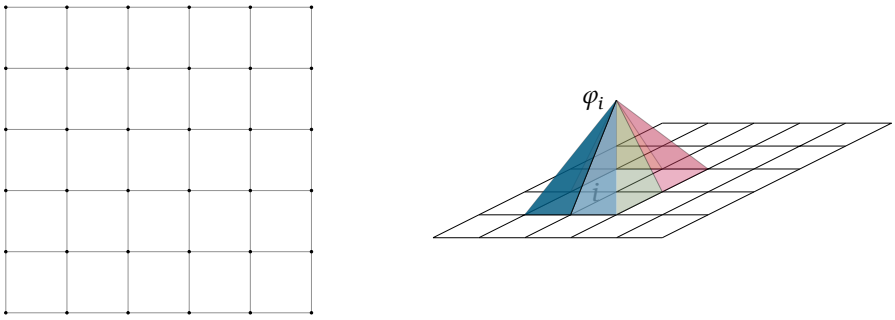


Figure 3.1: Finite Element space. Left: Mesh consisted of squares. Right: Linear test function.

where $U_j, j = 1, \dots, N$ are to be determined. In order to (3.5) hold for every $v_h \in V_h$ it is sufficient that it holds for every basis function, so we get next formulation of problem (3.5): find $(U_1, \dots, U_N) \in \mathbb{R}^N$ such that

$$DU_j \sum_{j=1}^{N} \int_\Omega \nabla\varphi_j(x)\nabla\varphi_i(x)dx + \sum_{j=1}^{N} U_j \int_\Omega \varphi_j(x)\varphi_i(x)dx = \int_\Omega f(x)\varphi_i(x)dx, \quad \forall i = 1, \dots, N. \tag{3.5}$$

If we define matrices $\mathbf{A}$, $\mathbf{M}$ and vector $\mathbf{F}$ in a following way

$$\begin{aligned}
\mathbf{A} &= (a_{ij}), a_{ij} = \int_\Omega \nabla\varphi_j(x)\nabla\varphi_i(x)dx, \quad i,j = 1, \dots, N \\
\mathbf{M} &= (m_{ij}), m_{ij} = \int_\Omega \varphi_j(x)\varphi_i(x)dx, \quad i,j = 1, \dots, N, \\
\mathbf{F} &= (f_i), f_i = \int_\Omega f(x)\varphi_i(x)dx, \quad i = 1 \ \dots \ N,
\end{aligned}$$

we see that $(U_1, \dots, U_N)$ is the solution of next linear system

$$(D\mathbf{A} + \mathbf{M}) \ \mathbf{U} = \mathbf{F}.$$

The next question is how will we choose space V_h. First we have to assume that Ω is a bounded domain with polygonal boundary $\partial\Omega$ so that it can be covered by a finite number of triangles or squares. It is assumed that any pair of triangles or squares intersect along a complete edge, at a vertex, or not at all, as shown in Figure 3.1a. With each interior node i we associate a basis function φ_i which is equal to 1 at node i and is equal to 0 at all the other nodes. An example of a test function is shown in Figure 3.1b.

The only thing left to discuss is how will we treat boundary conditions since we have the condition $u_i = 0$ if i is an exterior node. The easiest solution would be to put 0 on each entry of i^{th} row of matrix $\mathbf{A}$ except the diagonal one where we put 1. Also, we set all values of i^{th} row of matrix $\mathbf{M}$ to 0. That way we have assured that $u_i = 0$. In case of non-homogeneous Dirichlet conditions or Neumann conditions, the matrices $\mathbf{A}$ and $\mathbf{M}$ are easily modified. Case of periodic boundary conditions will be discussed later. This method is called the *Finite Element Method*. For more details see e.g. [8].

Now, let us go back to the Equation (3.1). We can write $U_{m+1}^{k+1}(x) = \sum_{j=1}^{N} U_{m+1,j}^{k+1}\varphi_j(x)$,

$b_m^k(x) = \sum_{j=1}^N b_{m,j}^k \varphi_j(x)$ and after multiplying with test functions we get

$$\sum_{j=1}^{N} U_{m+1,j}^{k+1} \left(\varphi_j, \varphi_i\right) + D\Delta t_m \sum_{j=1}^{N} U_{m+1,j}^{k+1} \left(\nabla\varphi_j, \nabla\varphi_i\right) = \sum_{j=1}^{N} b_{m,j}^k \left(\varphi_j, \varphi_i\right), \quad \forall i = 1 \ldots N.$$

If we define matrices **A** and **M** in a way described before we can see that the solution $U_{m+1}^{k+1}(x)$ of SDC sweep in $k+1^{\text{th}}$ iteration is the solution of the linear system

$$\mathbf{M}\mathbf{U}_{m+1}^{k+1} + \Delta t_m \mathbf{A}\mathbf{U}_{m+1}^{k+1} = \mathbf{M}\mathbf{b}_m^k \,. \tag{3.6}$$

4 DUNE AND PFASST++

DUNE, the "Distributed and Unified Numerics Environment" is a C++ library for solving partial differential equations with grid-based methods. It supports the easy implementation of methods like Finite Elements, Finite Volumes, and also Finite Differences. The framework consists of a number of modules which are downloadable as separate packages. In our implementation we only used the set of core modules. The development of DUNE started in 2002 and the main public representation of DUNE is its project homepage at `www.dune-project.org`.

The PFASST++ project given in [6] is a C++ implementation of the "parallel full approximation scheme in space and time" algorithm, which in turn is a time-parallel algorithm for solving ODEs and PDEs [2]. It also contains basic implementations of the spectral deferred correction (SDC) and multi-level spectral deferred correction (MLSDC) algorithms. The two parts have been combined to one application (dune-PFASST++) by Ruth Schöbel at Jülich Spercomputing Centre and it is still under development.

Since in each iteration of SDC method we have to solve linear system (3.6), here we describe the main idea behind the implementation of the Finite Element Method in DUNE. Let us suppose that we have constructed a mesh which covers the domain Ω. The mesh will be denoted by T_h. Furthermore, the number of vertices of each element $K \in T_h$ will be denoted by n. We associate two indices to each node: a local index $I \in \{1, \ldots, n\}$ and a global index $i \in \{1, \ldots, N\}$, where N is the dimension of the space V_h. It is important to construct a mapping from local to global indices

$$g : T_h \times \{1, \ldots, n\} \to \{1, \ldots, N\}.$$

Also, for each $K \in T_h$, $I, J \in \{1, \ldots, N\}$ we define a_{IJ}^K and m_{ij}^K in a following way

$$a_{IJ}^K = \int_K \nabla\varphi_J \nabla\varphi_I \,, \quad m_{IJ}^K = \int_K \varphi_J \varphi_I.$$

Now we have the next algorithm to assemble matrices **A** and **M**:

Algorithm 1 Algorithm for assembling matrices **A** and **M**

```
for all elements K of the mesh T_h do
  for all nodes I, J of K do
    calculate a_IJ^K and m_IJ^K
    get global index i of node I, i = g(I, K)
    get global index j of node J, j = g(J, K)
    a_ij = a_ij + a_IJ^K, m_ij = m_ij + m_IJ^K
  end for
end for
```

5 Results

In this section we aim to examine the performance of semi-implicit SDC scheme, first for a flame propagation problem and then for a linear multi-component problem. For both problems we know exact solutions. We also test the order of convergence in space and time.

5.1 Flame propagation problem

We consider the family of 1D reaction-diffusion equations

$$\frac{\partial u}{\partial t} = \Delta u + \frac{8}{\delta^2} u^2(1-u), \quad -\infty \le x \le \infty, \quad \delta > 0,$$

with boundary conditions $u(x) \to 1$ as $x \to -\infty$ and $u(x) \to 0$ as $x \to \infty$. The exact solution is given by

$$u(x,t) = \frac{1}{2}\left(1 - \tanh\left[\frac{x-ct}{\delta}\right]\right), \quad c = \frac{2}{\delta}. \tag{5.1}$$

For numerical simulations we used a finite element grid over the interval $[-20, 20]$, $\delta = 1$ and Dirichlet condition: $u(-20,t) = 1$, $u(20,t) = 0$. For time-step discretization we used 3 Gauss-Radau nodes and, as expected, we got 5^{th} order convergence in time, see Figure 5.1.

5.2 Linear multi-component model

Here we consider a one dimensional reaction-diffusion system with two state variables

$$\begin{aligned} \frac{\partial u}{\partial t} &= d\Delta u - au + v, \\ \frac{\partial v}{\partial t} &= d\Delta v - bv, \end{aligned} \tag{5.2}$$

with $x \in [0, \pi/2]$ and boundary conditions:

$$\frac{\partial u}{\partial x}(0,t) = 0, \quad \frac{\partial v}{\partial x}(0,t) = 0, \quad u(\frac{\pi}{2},t) = 0, \quad v(\frac{\pi}{2},t) = 0.$$

The diffusion parameter d and the reaction parameters a and b are fixed. The system admits the following family of solutions:

$$\begin{aligned} u(x,t) &= \left(e^{-(a+d)t} + e^{-(b+d)t}\right)\cos(x) \\ v(x,t) &= (a-b)e^{-(b+d)t}\cos(x). \end{aligned}$$

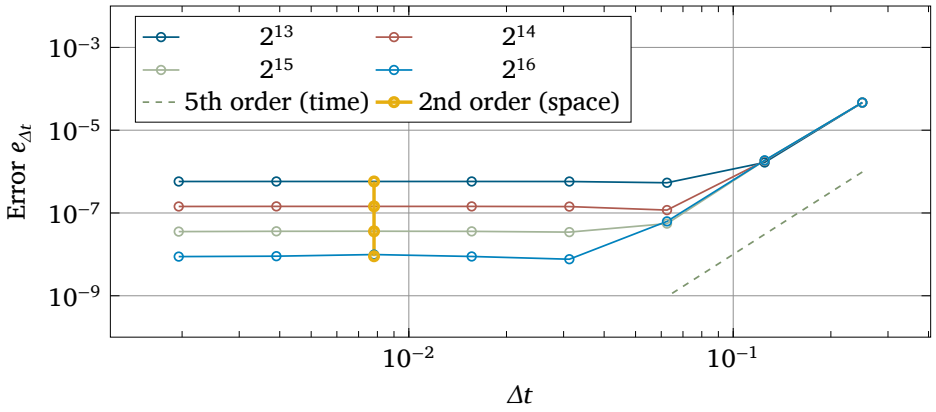


Figure 5.1: Log-log plot of error in dependence of time for different number of finite elements.

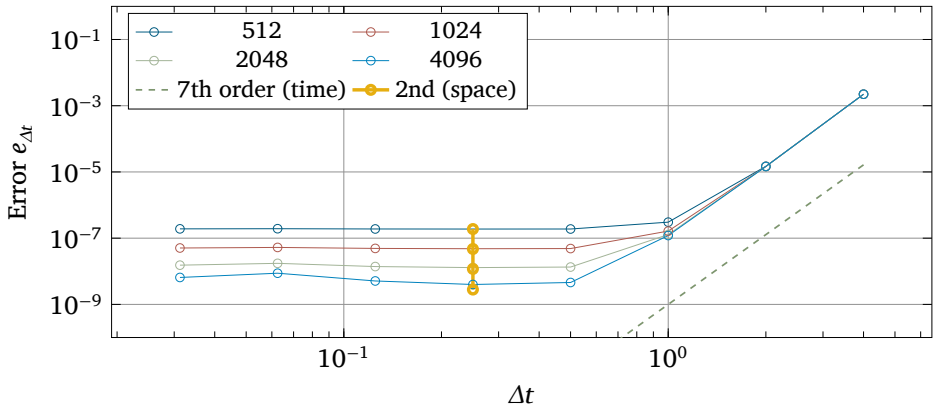


Figure 5.2: Log-log plot of error in dependence of time for different number of finite elements; $a = 0.1, b = 0.01, d = 1$.

In this example we used 4 Gauss-Radau nodes and tested for diffusion dominant case with the triplet $(a, b, d) = (0.1, 0.01, 1)$. Figure 5.2 shows 7^{th} order convergence in time and 2^{nd} order convergence in space.

For the implementation in DUNE we note that the system (5.2) can also be written as

$$\frac{\partial \mathbf{U}}{\partial t} = D\Delta\mathbf{U} + B\mathbf{U}\,, \tag{5.3}$$

where $\mathbf{U} \in \mathbb{R}^2$, $\mathbf{U}_1 = u$, $\mathbf{U}_2 = v$, and matrices D, $B \in \mathbb{R}^{2\times 2}$ are defined in a following way

$$D = \begin{bmatrix} d & 0 \\ 0 & d \end{bmatrix}, \qquad B = \begin{bmatrix} -a & 1 \\ 0 & -b \end{bmatrix}.$$

We change the implementation in a way that the unknown variable u is now a vector of block-vectors instead of doubles. Each block-vector has the size two. If we had N

components in our model, we would have a block vector of size N. Also, we had to change structure of the matrices $\mathbf{A}$ and $\mathbf{M}$. We defined their entries to be 2×2 matrices so that it would be possible to apply them on vector $\mathbf{U}$. In other words, we define new matrices in the multicomponent model as $\mathbf{A}_{multicomp} := \mathbf{A} \otimes \mathbf{I}_2$, $\mathbf{M}_{multicomp} := \mathbf{M} \otimes \mathbf{I}_2$. This way we generalized the implementation in dune-PFASST++ to support also system of reaction-diffusion equations.

6 Gray-Scott model

The Gray-Scott model is a reaction-diffusion system that involves two generic chemical species U and V, whose concentration at a given point in space are denoted by u and v. They react with each other, they diffuse through the medium and V produces an inert product P. Therefore, the concentration of U and V at any given location changes with time and can differ from that at other locations. There are two reactions which occur at different rates throughout the space according to the relative concentrations at each point:

$$\begin{aligned} U + 2V &\rightarrow 3V, \\ V &\rightarrow P\,. \end{aligned}$$

The overall behavior of the system is described by the formula

$$\begin{aligned} \frac{\partial u}{\partial t} &= D_u \Delta u - uv^2 + F(1-u) \\ \frac{\partial v}{\partial t} &= D_v \Delta v + uv^2 - (F+K)v. \end{aligned}$$

The first equation tells how quickly the quantity u increases. The first term, $D_u \Delta u$ is the diffusion term. It specifies that u will increase in proportion to the Laplacian. The second term is -uv^2. This is the reaction rate. The first reaction written above requires one U and two V; such a reaction takes place at a rate proportional to the concentration of U times the square of the concentration of V. Also, it converts U into V: the increase in v is equal to the decrease in u. The third term, $F(1-u)$, is the replenishment term. Since the reaction uses up U and generates V, all of the chemical U will eventually get used up unless there is a way to replenish it. The replenishment term says that u will be increased at a rate proportional to the difference between its current level and 1. As a result, even if the other two terms had no effect, 1 would be the maximum value for u. The constant F is the feed rate and represents the rate of replenishment. The third term in the v equation is the diminishment term. Without the diminishment term, the concentration of V could increase without limit. For more information on this chemical system see [5] and [3].

We wanted to test our implementation of this multi-component reaction-diffusion system for different values of F and K. Also, since we wanted to simulate a large tank without having a big domain, we implemented periodic boundary conditions.

Now we present implementation of matrices $\mathbf{A}$ and $\mathbf{M}$ which corresponds to periodic boundary conditions. First, for simplicity, lets take the $1D$ domain $[0,1]$. Let $\varphi_1, \dots, \varphi_N$

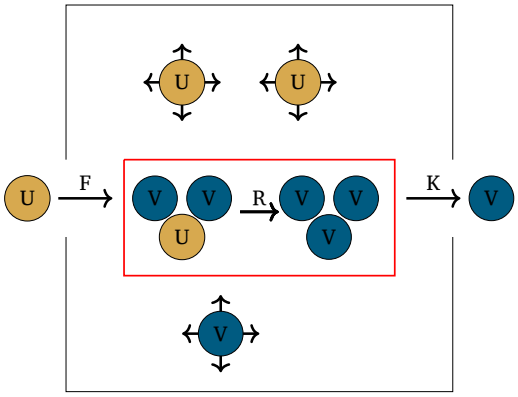


Figure 6.1: Illustration of the reaction diffusion model. V is fed at a certain rate, while U is removed ('killed') at a certain rate. Both compound U and V diffuse at certain rates, and U reacts with $2V$ and is converted into V.

be the test functions. Taking in account that we want $u(0) = u(1)$, our solution u can be written as

$$u(x) = u_1\varphi_1 + u_2\varphi_2 + \ldots + u_1\varphi_N. \tag{6.1}$$

Let us define test functions ψ_i, for $i = 1, 1, \ldots, N-1$ as

$$\begin{aligned} \psi_1 &= \varphi_1 + \varphi_N \\ \psi_i &= \varphi_i, \quad i = 2, \ldots, N-1. \end{aligned}$$

Now, since $u_N = u_1$ the next formula holds for solution u:

$$u(x) = u_1\psi_1 + u_2\psi_2 + \ldots + u_{N-1}\psi_{N-1}.$$

So, we got one degree of freedom less than we had before, that is, $V_h = [\{\psi_1, \ldots, \psi_{N-1}\}]$. We define matrices $\mathbf{A}_p$ and $\mathbf{M}_p$ as described in Section 3 as

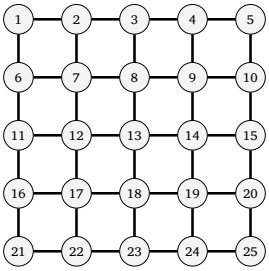
$$\begin{aligned} A_p &= (a_{ij}^p), \quad a_{ij}^p = \int_0^1 \nabla\psi_j \nabla\psi_i, \\ M_p &= (m_{ij}^p), \quad m_{ij}^p = \int_0^1 \psi_j\psi_i. \end{aligned}$$

The function ψ_1 shares its support with functions ψ_2 and ψ_{N-1}. That leads to the conclusion that in the first row of matrices $\mathbf{A}_p$ and $\mathbf{M}_p$ the non-zero elements would be $(1,1), (1,2)$ and $(1, N-1)$. So, now we can easily reproduce matrices $\mathbf{M}_p$ and $\mathbf{A}_p$ from

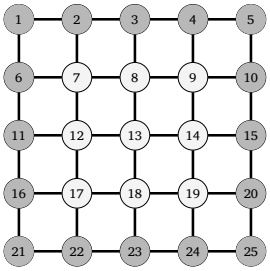
matrices **M** and **A**:

$$\mathbf{M}_p = \begin{bmatrix} 2m_{11} & m_{12} & 0 & \dots & m_{N,N-1} & 0 \\ m_{21} & m_{22} & m_{23} & 0 & \dots & 0 \\ 0 & & \ddots & & & \\ \vdots & & & \ddots & & \\ \vdots & & & & \ddots & \\ 0 & & & & & 0 \end{bmatrix}, \quad \mathbf{A}_p = \begin{bmatrix} 2a_{11} & a_{12} & 0 & \dots & a_{N,N-1} & 0 \\ a_{21} & a_{22} & a_{23} & \dots & & \\ \vdots & & \ddots & & & \\ \vdots & & & \ddots & & \\ \vdots & & & & \ddots & \\ 1 & & & & & -1 \end{bmatrix}$$

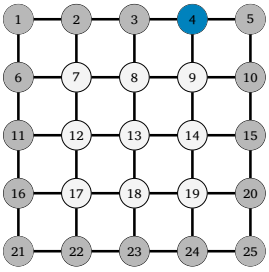
We integrated condition $u_1 = u_N$ into the last row of matrices $\mathbf{A}_p$ and $\mathbf{M}_p$. Generalization for the $2D$ domains is shown in the Figure below.



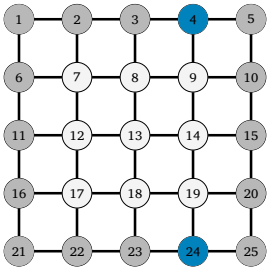
(a) First we have only a vector of global indices.



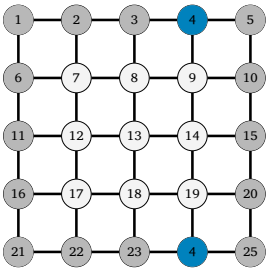
(b) We distinguish boundary nodes from the interior nodes.



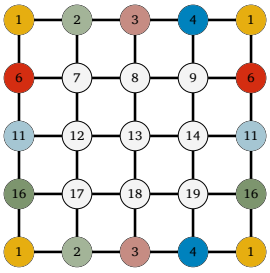
(c) We locate node 4 on the boundary.



(d) We find its periodic pair – node 24.



(e) In the vector of global indices we change its global index to 4.



(f) We do the same procedure for all boundary nodes.

Figure 6.2: This set of pictures depicts the implementation of periodic boundary conditions in DUNE for 2*D* domains. The idea can be extended for 3*D* domains.

7 Numerical simulation

Simulations are performed on a finite domain $[0,1] \times [0,1]$ with 100 finite elements in each direction. Different values for parameters F and K are considered.

We can see that the numerical simulation is very sensitive to the parameters F and K. In both simulations we had the same initial condition and same timestep size $dt = 10s$. In the first simulation, as stable state we got circles (see Figure 7.1) and in the second simulation shown in Figure 7.2 as stable state we got stripes. Also, what can be seen nicely in the simulation shown in Figure 7.1 is a process similar to mitosis – at convex borders more U is available to diffuse inwards and feed V, so those edges grow outwards as V increases and diffuses. At concave borders, less U diffuses in and V slowly thins out. This causes a spot of V to grow and then divide into two as if it were a cell undergoing mitosis.

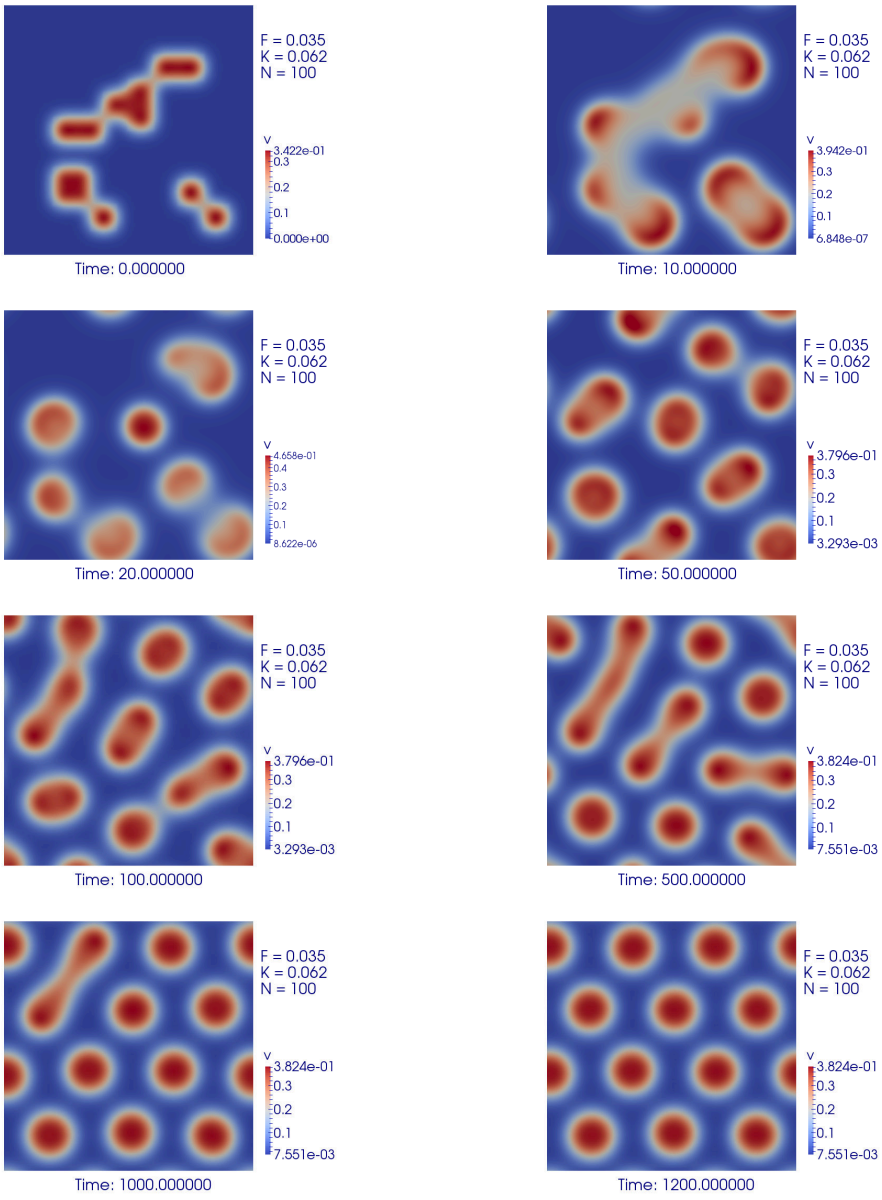


Figure 7.1: Numerical simulation for parameters $F = 0.035$, $K = 0.062$, $dt = 10s$

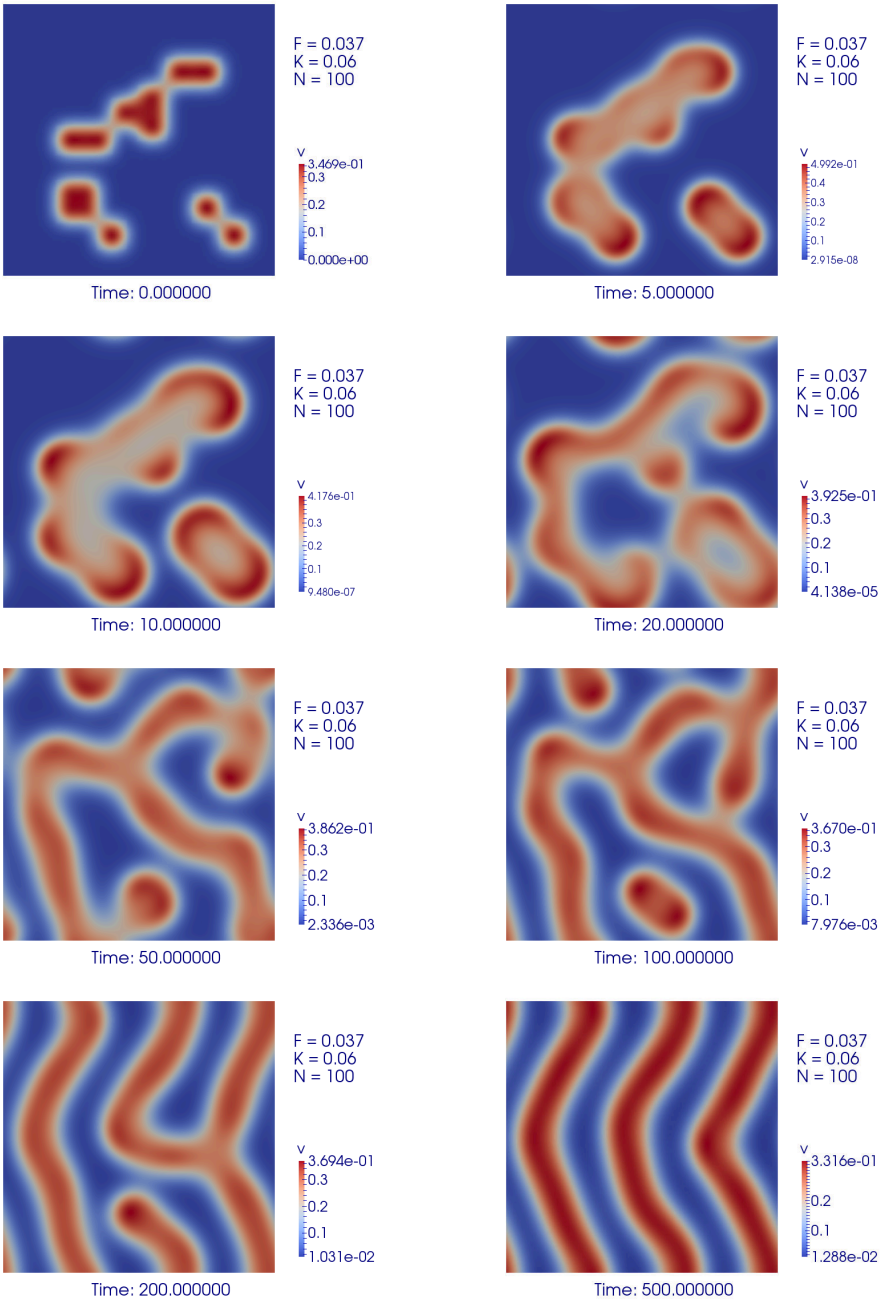


Figure 7.2: Numerical simulation for parameters $F = 0.037$, $K = 0.060$, $dt = 10s$

8 Conclusion

In this paper we present results of the implementation of the multi-component reaction diffusion systems in dune-PFASST++. First we extended the existing implementation of a solver for the heat equation to support also a reaction term and then we implemented a multi-component reaction-diffusion solver. We examined the performance through numerical examples such as a flame propagation problem and linear multi-component problem for which we know exact solutions. Results have shown good behaviour in terms of space and time order of convergence. In order to get numerical simulations of the Gray-Scott model we also implemented periodic boundary conditions. We ran simulation for different F and K and concluded that the model is very sensitive to changing the parameters. The next step of this project would be to parallelize in the temporal domain and fully use features of the PFASST++ library, such as MLSDC. In contrast to SDC, MLSDC performs correction sweeps in time on a hierarchy of discretization levels. The advantage of MLSDC is that it shifts computational work from the fine level to coarse levels, thereby reducing the number of fine SDC sweeps and, therefore, the time-to-solution. The method is described in detail in [7].

9 Acknowledgements

First I want to thank Ivo Kabadshow for organising this programme and choosing me to be part of it. Secondly, I want to thank my colleagues who I spent my last two months with. I carry you all in my heart. Finally, I would like to express my gratitude to my advisors Dr. Robert Speck and Ruth Schöbel for their help and support during this programme.

References

[1] A. Dutt, L. Greengard, and V. Rokhlin. **Spectral deferred correction methods for ordinary differential equations**. *BIT Numerical Mathematics*, 40(2):241–266, 2000.

[2] M. Emmett and M. Minion. **Toward an efficient parallel in time method for partial differential equations**. *Communications in Applied Mathematics and Computational Science*, 7(1):105–132, 2012.

[3] K. J. Lee, W. McCormick, Q. Ouyang, and H. L. Swinney. **Pattern formation by interacting chemical fronts**. *SCIENCE-NEW YORK THEN WASHINGTON-*, 261:192–192, 1993.

[4] M. L. Minion et al. **Semi-implicit spectral deferred correction methods for ordinary differential equations**. *Communications in Mathematical Sciences*, 1(3):471–500, 2003.

[5] J. E. Pearson. **Complex patterns in a simple system**. *arXiv preprint patt-sol/9304003*, 1993.

[6] **http://www.parallelintime.org/pfasst/**. `http://www.parallelintime.org/PFASST/`.

[7] R. Speck, D. Ruprecht, M. Emmett, M. Minion, M. Bolten, and R. Krause. **A multi-level spectral deferred correction method**. *BIT Numerical Mathematics*, 55(3):843–867, 2015.

[8] E. Süli. **Lecture notes on finite element methods for partial differential equations**. *Mathematical Institute, University of Oxford*, 2012.

Alternative Communication Methods for Stencil-Based Operations in MPI

Lukas Mazur
Department of Physics
Bielefeld University
Germany
lmazur@physik.uni-bielefeld.de

Abstract *Most simulations in Quantum Chromodynamics (QCD) follow a specific communication pattern. We present an implementation which is using one sided communication. First, we explain in detail how the communication pattern can be implemented with two sided communication calls. Then we show how to improve this pattern using one sided communication and by introducing double buffering. In the end we present measurements which show the performance differeces between both methods.*

1 Introduction

Quantum Chromodynamics (QCD) is a theory of quarks and gluons and thus of all hadronic matter. Before 1970 the only way to study QCD was by using a pertubative expansion at small distances or high temperatures. 1974 Kenneth Wilson discretized the theory by replacing space-time with a four dimensional lattice [3]. The first numerical simulations where done by Michael Creutz in 1980 [2]. Since then lattice QCD became a well-established non-perturbative approach to solving QCD problems.

Today most simulations in this field require a lot of computation time. Therefore, the massive increase of computation power and better algorithms have opened new fields of study. In this paper we try to improve a common used algorithm in Lattice QCD. We present a different communication method for stencil operations in MPI [1].

First we explain in section 2 what we denote as a stencil. In section 3 we show how to create an cartesian grid in MPI. Then we describe a commonly used implementation in section 4. In section 5 we introduce the most important one sided communication calls in MPI and explain why we should benefit from using them. Subsequently, in section 6 we introduce the “double buffering” method. Our implementation of these approaches is shown in section 7. Finally we present our performance results in section 8 and conclude them in section 9.

2 Stencil

A stencil can be understood as a geometric arrangement that introduces a relationship between a point and its neighbours using a numerical approximation routine. In Lattice QCD it is a partial differential equation. Figure 2.1 visualizes a stencil on a two dimensional lattice. In order to parallelize it, one can domain-decompose the lattice into sub-lattices equal to the number of parallel processes. Then the stencil computations are done on each sub-lattice in parallel. Within one sub-lattice the computation only requires data from lattice points associated with that process and thus no data need to be transferred between processes at these points. However, the lattice points on the surface of each sub-lattice require information from other ranks. In our work we are only focussing on interprocess communication, thus each lattice point is associated with

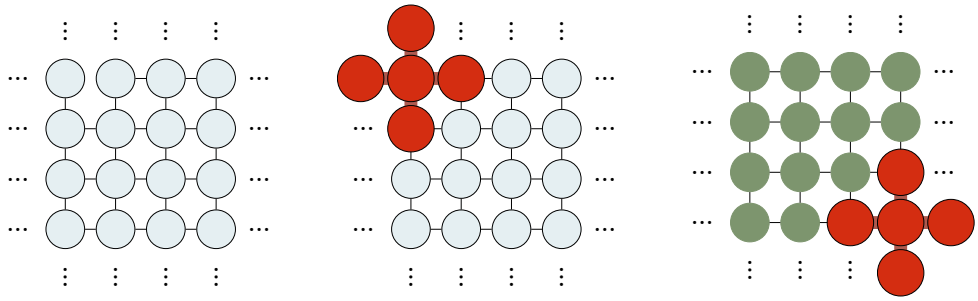


Figure 2.1: QCD Lattice. Each circle holds a physical value which needs to be updated. This is done via the stencil (red) which goes through all lattice points. After one sweep is done, the stencil starts again from the beginning.

one process. After one sweep through the whole lattice, the computation starts again from the beginning updating the new calculated values on each lattice point. A lot of iterations in Lattice QCD usually are done in the same manner, which leads to long computation times.

3 Lattice in MPI

Lattice QCD uses a four dimensional (space-time) lattice. Since the stencil only connects nearest neighbours, we can domain-decompose the lattice with the implication that only neighbouring compute nodes exchange data. In order to implement this, we first need to rearrange the ranks in a way that each rank knows its neighbours. This can be done by `MPI_Cart_create`:

Listing 12.1: MPI_Cart_create Interface

```
1 int MPI_Cart_create(MPI_Comm comm_old, int ndims,
2         const int dims[], const int periods[],
3         int reorder, MPI_Comm *comm_cart)
```

This function needs an MPI communicator, the number of dimensions, the size of each dimension, an array which indicates which dimension should be periodic and the integer `reorder` which specifies whether the ranks should be relabeled or not. It returns a new communicator, which has the desired properties as shown in figure 3.1. If a rank wants to know which neighbours he has, one should use the function `MPI_Cart_shift`:

Listing 12.2: MPI_Cart_shift Interface

```
1 int MPI_Cart_shift(MPI_Comm comm, int direction,
2         int disp, int *rank_source, int *rank_dest)
```

That function takes as arguments the cartesian communicator, the direction which indicates the dimension of interest and a displacement where the distance to the neighbour is specified (e.g. nearest neighbour $= 1$, next nearest neighbour $= 2$). It returns the two ranks `rank_source` and `rank_dest` which are the neighbours in the given dimension.

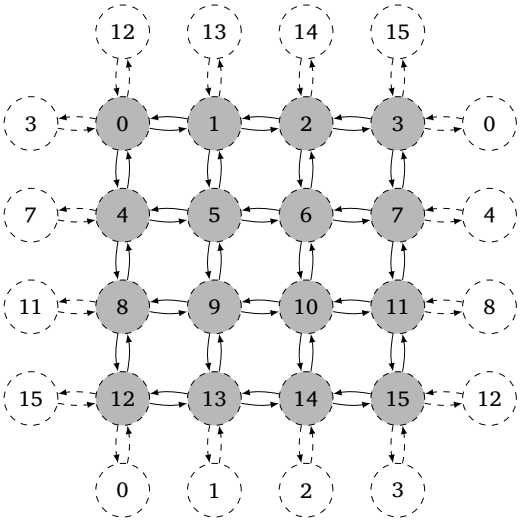


Figure 3.1: 2D 4x4 cartesian communication in MPI. Each number represents a rank. Optionally one can also turn on periodic boundary conditions, as shown with dashed circles.

4 Simple communication pattern

In this section we want to illustrate a rather simple communication pattern. Usually MPI does not guarantee that processes are running synchronously. Therefore, the programmer has to take care about synchronization. A commonly used stencil algorithm would go through the following steps:

1. Send my local values to my neighbours
2. Calculate a new local value out of my old value and the received values of my neighbours
3. Wait until neighbours are done with the same iteration step, before sending the new calculated value to them.

In MPI such a pattern can easily be implemented as follows:

Listing 12.3: Implementation

```
1  for(int i= 0; i<maxupdates; i++){
2
3     MPI_Irecv(right_recv, 1, buffer_type, right, ... );
4     MPI_Irecv( left_recv, 1, buffer_type,  left, ... );
5     MPI_Irecv(   up_recv, 1, buffer_type,    up, ... );
6     MPI_Irecv( down_recv, 1, buffer_type,  down, ... );
7
8     MPI_Isend(sendbuf, 1, buffer_type,  left, ... );
9     MPI_Isend(sendbuf, 1, buffer_type, right, ... );
10    MPI_Isend(sendbuf, 1, buffer_type,  down, ... );
11    MPI_Isend(sendbuf, 1, buffer_type,    up, ... );
12
13    MPI_Waitall(4, request_send, ... );
14    MPI_Waitall(4, request_recv, ... );
15
16    update_buffer(left_recv, right_recv, ... , sendbuf);
```

Each rank enters this loop, which means that each iteration step represents a whole sweep through the lattice. The `MPI_Irecv` calls are for receiving data from neighbours, while the `MPI_Isend` calls are sending the local buffer to the neighbours. These calls are non-blocking calls, which means that they return immediately. Therefore, one has to make sure that the communication is done by using `MPI_Waitall`. Afterwards, the sendbuffer will be updated using the received data from the neighbours. It is important to wait not only for `MPI_Irecv` but also for `MPI_Isend`, because otherwise the sendbuffer will be overwritten by `update_buffer` while still in use by the sending calls (One could avoid this by introducing a second sendbuffer). These MPI functions ensure that each `MPI_Isend` call has only one corresponding `MPI_Irecv` call. That means, if one rank is one iteration step further than one of his neighbours, he can not send his new computed buffer to him, because the neighbour did not call the corresponding `MPI_Irecv` function so far.

5 One sided communication

So far, we just used two sided communication calls. Hence both processes are involved in communication as shown in figure 5.1a. If one rank wants to send something to another rank, then the sending rank has to send a sending request to the receiving rank first. After that, the sending rank has to wait until the receiving rank sends a so called “handshake”. As soon as the receiving rank did that, the sending rank is allowed to send his data to the receiving rank. In theory these handshakes are causing more communication overhead. To avoid this, one could use one sided communication calls as shown in figure 5.1b. With these calls, only one rank is involved in communication. Hence, if one rank wants to send something to another rank, it just puts its data into the memory of the other rank without waiting for a handshake. The effect should be seen when we measure how long the program needs to get through all iterations. In theory the one sided version should finish earlier than the two sided version as shown in figure 5.2, because the ranks do not need to wait for handshakes anymore. This effect should become clearer when we increase the iteration count. In MPI the corresponding one sided communication call would be `MPI_Put`:

Listing 12.4: MPI_Put Interface

```
1 int MPI_Put(const void *origin_addr, int origin_count,
2          MPI_Datatype origin_datatype, int target_rank,
3          MPI_Aint target_disp, int target_count,
4          MPI_Datatype target_datatype, MPI_Win win)
```

Among other arguments this function essentially takes the sendbuffer, the target rank, the displacement in the target buffer and the window as arguments. A window object is created with `MPI_Win_allocate` and specifies a “window” in the memory of a process that is made accessible for access by remote processes. `MPI_Put` can only be called within a so called “epoch”. In our work we are using a passive target epoch for all remote memory access (rma) calls. A passive target epoch starts with `MPI_Win_lock` and ends with `MPI_Win_unlock`.

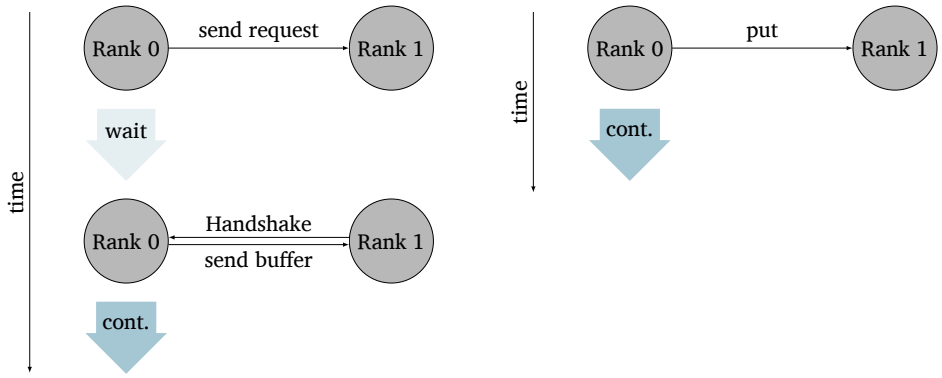


Figure 5.1: Two sided communication (left) compared to one sided communication (right). Circles indicate ranks, while thin arrows represent communication. One can see that two sided communication needs more communication calls than one sided communication.

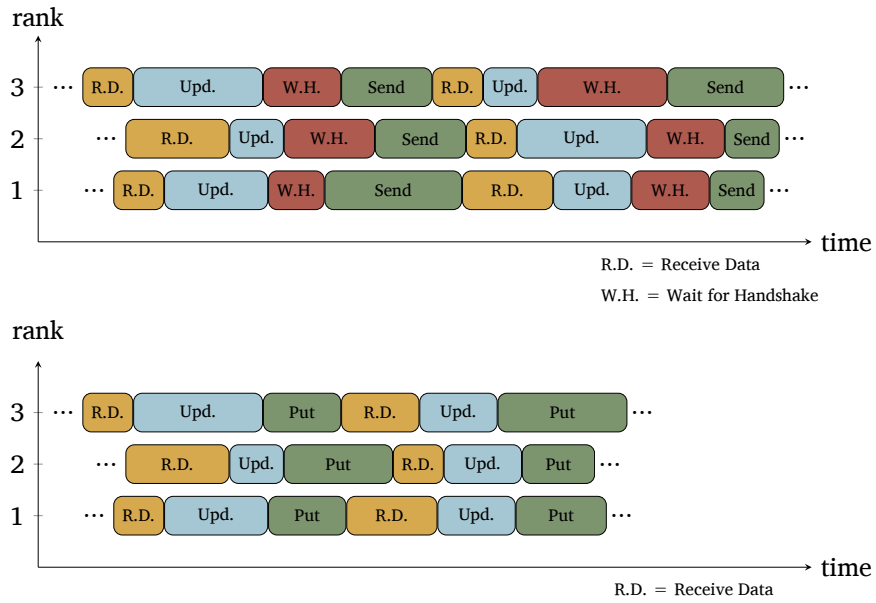


Figure 5.2: Schematic timeline of three ranks performing two iterations. The picture above represents a two sided communication pattern while the picture below shows a one sided communication pattern. Since the waiting times in the second picture disappear, the second method should finish earlier.

6 Double buffered communication

The communication pattern described in section 4 is not really efficient. The reason is that in step 3 the process is wasting communication time. The process can not communicate the new buffer and has to wait for the neighbours. To avoid this, we can introduce a second receivebuffer. Then the communication pattern would look like that:

1. Send my local values to my neighbours
2. Calculate new local value out of my old value and the received values of my neighbours
3. Send the new calculated value to the second receivebuffer of my neighbours
4. continue computing

Using that pattern the communication step for the new calculated value is already done, which means that the process does not need to wait for the neighbours before sending the new buffer. Figure 6.1 visualizes the difference between the single buffer approach (left) and the double buffer approach (right). In the single buffer visualization, each rank has only one receivebuffer per neighbour, whereas in the double buffer visualization, each rank has two receivebuffers respectively. In figure 6.1a some ranks have already exchanged some data, thus rank 1 got all data needed to compute the new data out of the receiving- and sendingbuffer. This is shown in figure 6.1b, where the dark blue filled rectangle denotes the new data. Meanwhile rank 0 and 2 did not get all required information. Figure 6.1c shows the advantage of the double buffer approach. In the single buffer approach, rank 1 can not send its sendbuffer to rank 2, because there is still data in the receiving slot of rank 2, which is required for further calculation. Therefore, rank 1 has to hold on communication, until rank 2 is done. In the double buffer approach instead, rank 1 does not need to wait and can send its buffer immediately to the second buffer of rank 2.

Figure 6.1: Single buffered method (left) compared to double buffered method (right). Within one dimension three ranks are communicating with each other (thin arrows). The big rectangles with rounded corners represent the memory of each rank, while the smaller rectangles inside represent the communication buffer. The single upper rectangle stands for the sending buffer, while the lower rectangles represent receiving buffers. White filled rectangles denote empty buffers, whereas colored rectangles represent buffers with valid data.

7 IMPLEMENTATION

In MPI a double buffered communication algorithm with one sided communication calls can be implemented as follows:

LISTING 12.5: IMPLEMENTATION

```
for(int i= 0; i<maxupdates; i+=2){
    MPI_Put(sendbuf, 1, buffer_type,  right,   left_displ, ... , win);
    MPI_Put(sendbuf, 1, buffer_type,   left,  right_displ, ... , win);
    ...
    MPI_Win_flush_local_all(win);
    ... compute something ...
    while(1){
        MPI_Iprobe(cart_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
        MPI_Win_sync(win);
        if(left_recv[length] == i && ... top_recv[length] == i) break;
    }
    update_buffer(left_recv, right_recv, ... , sendbuf, i+1);

    MPI_Put(sendbuf, 1, buffer_type,  right,   left_displ, ... , win2);
    MPI_Put(sendbuf, 1, buffer_type,   left,  right_displ, ... , win2);
    ...
    MPI_Win_flush_local_all(win2);
    ... compute something ...
    while(1){
        MPI_Iprobe(cart_rank, MPI_ANY_TAG, MPI_COMM_WORLD, &flag, &status);
        MPI_Win_sync(win2);
        if(left_recv2[length] == i+1 && ... top_recv2[length] == i+1) break;
    }
    update_buffer(left_recv2, right_recv2, ... , sendbuf, i+2);
```

In this example, the step size in the `for` loop is not 1 anymore but 2. This is due to the double buffering method, because we are fusing two iterations. After the first `MPI_Put` calls we need to call `MPI_Win_flush_local_all` in order to make sure that the `MPI_Put` operations are locally done. That does not necessarily mean that the target has already received this data. But for our purposes we do not need to know this. As soon as everything is flushed locally, we can compute something to overlap communication with computation. The `while` loop afterwards checks if the receivebuffer holds new data. To do this we should always synchronize public memory with private memory before, since some systems are using a so called RMA seperate model. In that model, there is only one instance of each variable in process memory, but a distinct public copy of the variable for each window that contains it. Additionally one has to call `MPI_Iprobe` to advance communication, otherwise some ranks would not return from `MPI_Win_flush_local_all`. `MPI_Put` does not need a corresponding receive call like `MPI_Isend`. That means that we also have to send information about the status of the receivebuffer. If we would not do that, then the receiving rank would not know whether it can proceed with that buffer or not. In our case we send an additional integer value together with our data, which indicates in which iteration step this data was computed. Then the `if` statement in the `while` loop checks for the additional integer. When the `while` loop returns, the sendbuffer can be updated by using the receivebuffer. After this is done, the rank can immediatly continue with sending the new computed data to the second receivebuffer of the target. This is done by using a different communication window (`win2`).

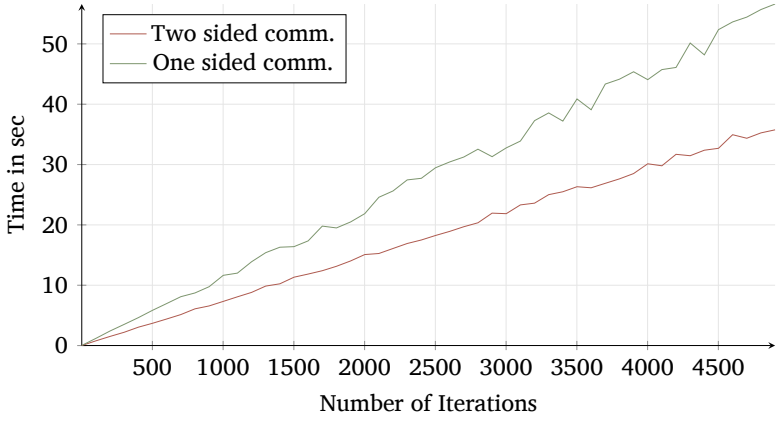


Figure 8.1: Two sided and one sided communication in comparison.

8 Results

To measure the difference between the two sided version and the double buffered one sided version, we measured the time required for both programs to finish different counts of iterations. The x-axis in figure 8.1 refers to the maximum iteration count and the y-axis shows how long it takes to compute that number of iterations. We tested our programs on 8 JURECA nodes with one core per node. Each node holds two Intel Xeon E5-2680 v3 Haswell CPUs with a clock rate of 2.5GHz. The nodes are connected to each other via Mellanox EDR InfiniBand. As MPI implementation we are using IntelMPI version 5.1.3.181. The reason why we are choosing only one core per node is that we only want to measure the performance of internode communication. For intranode communication there are better approaches available than MPI.

Figure 8.1 shows that our one sided program is slower than the two sided version. Since in theory our approach has less communication overhead, we were expecting that the one sided line would be flatter than the two sided line. The reason for the bad performance is probably due to the details of the MPI implementation. Most of the MPI programs today are using only two sided calls. Therefore, the implementations are not optimized for one sided communication. Moreover, we do not know how the two sided calls in MPI are implemented. Even though in theory the two sided functions should follow the “handshake” protocol described in section 4, the implementation could handle it completely differently.

9 Conclusion

In this work we compared two communication methods for stencil based operations on a lattice. The first method is based on simple two sided communication calls and uses only one receivebuffer for each neighbour. The second method is based on one sided communication calls and uses two receivebuffers. Since the second method should have in theory less communication overhead and less waiting times, we expected that this method should be faster than the first method. As it turns out, the second method is slower than the first one. This is probably due to the MPI implementation, which is

not fully optimized for one sided communication calls. This reminds us once again that MPI is just a standard and not an implementation. One should also test this program on different MPI Implementations like MPICH, OpenMPI, etc. Furthermore, the double buffering method can also be tested using other two sided communication calls. As a final remark, we conclude that using one sided communication is (with the given implementation) not worth the effort. One has to mind a lot of pitfalls and in the end, the program is not faster. As long as the implementation is not optimized for one sided communications, one should stay with two sided communication calls.

References

[1] Message Passing Interface Forum. *MPI: a Message-passing Interface Standard: Version 3.1*. High-Performance Computing Center, 2015. https://books.google.de/books?id=uv1ajwEACAAJ.

[2] H. Suganuma, T. Iritani, F. Okiharu, T. T. Takahashi, and A. Yamamoto. **Lattice QCD Study for Confinement in Hadrons**. In A. Hosaka, K. Khemchandani, H. Nagahiro, and K. Nawa, editors, *American Institute of Physics Conference Series*, volume 1388 of *American Institute of Physics Conference Series*, pages 195–201, October 2011. arXiv:1103.4015, doi:10.1063/1.3647373.

[3] D. Xhako, R. Zeqirllari, and A. Boriçi. *Using Parallel Computing to Calculate Static Interquark Potential in LQCD*, pages 23–30. Springer International Publishing, Cham, 2014. http://dx.doi.org/10.1007/978-3-319-01520-0_3, doi:10.1007/978-3-319-01520-0_3.

A Violin Plot Plug-in for Cube
Performance Analysis With Style

Robert Poenaru
B. Sc. Physics
University of Bucharest
Romania
robert.poenaru@
protonmail.ch

Abstract *Cube is an open source software used for displaying performance data of HPC applications. Extending the toolset of Cube with an additional plug-in which is capable of making violin plots for numerical data sets will be of great help for the HPC community dealing with performance analysis. In this paper, a step-by-step description of the Violin Plot Plug-in is made, starting from the mathematical concept of a violin plot, up to the improvements on the algorithm as well as performance measurements of the algorithm.*

1 Introduction

Nowadays, studies in topics like Computer Science, Physics, Biology, Medicine etc., generate complex problems that often require simulations which can become too complex to run them on a *desktop environment*. Fortunately, there is an increasing number of HPC centres which can run these simulations, helping in this way the scientific community. Even though a supercomputer is a powerful machine, capable of computing speed which surpasses any modern PC, its internal structure is so complex, that the developers of any application must write the code in such a way that the application could use that computing power as much as possible. All the applications which are meant to be executed on a supercomputer require a parallel behaviour, which means that all the processors are executing calls at any given moment. Any serialization or inefficient routines in the code will make the running time increase, making a supercomputer to perform slower than its normal capabilities. Because making an application capable of running on large-scale computer systems is not an easy task, HPC community come with an affordable solution: *Performance Analysis*. This step assures the developer that his application takes advantage of (more or less) the entire computing power of the machine. This process require a preliminary measurement of the *unoptimized application*, then based on the measurement data, an in-depth analysis which will give options for further improvement/tuning of the application (see figure 1.1).

From this workflow, one can observe that performance analysis is an iterative process, which means that it can be performed multiple times on the same application, until it meets the desired degree of optimization.

However, in order to make performance analysis, the HPC community needs some tools capable of making the optimization of parallel applications both more effective and more efficient. Many research centres developed their own tools for optimizing applications; from tools which measure the performance of an application or search for inefficiencies in the application up to programs which display the measurement data. Performance analysis tools will help the developer to find the root causes of potential bottlenecks, serialization or any other inefficient behaviour in the application. Tuning the code in order to avoid such problems will increase the scalability of the application.

The only issue which arises after the performance measurement has been done is that the reports contain information about the entire code. Is not hard to realise that, for a

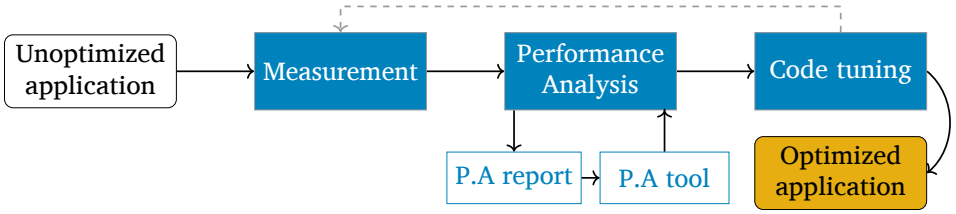


Figure 1.1: Scheme for developing an optimized algorithm.

complex algorithm, such reports can be very hard to analyse by the developers due to the enormous data. So, they need methods for quick data visualization, with a summarized view on the program behaviour. Cube is a software which is displaying the performance reports in a user-friendly way and aggregated information of the HPC application can be visualized in a scalable fashion. Cube has several plug-ins which extend user capabilities to interpret the numerical data. Examples of such plug-ins: *Topology, SystemBoxPlot.* In this paper, the description of a new plug-in is realised: *Violin Plot.* With this new tool, one can see the graphical representation of the distribution of data from the performance report. The remainder of this article is organized as follows: a short overview about performance analysis tools, an introduction into Cube software together with description of SystemBoxPlot plug-in. The next discussion is about the actual project: Violin plot; with theoretical background and algorithm implementation into Cube. Three methods for computing the violin plot are represented with performance results for each version. Finally the user interface of the plug-in is shown followed by conclusions and future work.

2 Performance analysis tools

Developers of parallel application can choose from a variety of performance-analysis tools. According to [2], these tools can be categorized into the following way:

On-line Provides immediate feedback on the performance behaviour while the application is still running, allowing the user to intervene if necessary.

- Periscope (Technische Universität München, Germany)
- Paradyn (University of Wisconsin, USA)

Post-mortem They record performance behaviour at runtime and then analyse it after the target application has terminated.

- Scalasca (Forschungszentrum Jülich & German Research School for Simulation Sciences – Aachen, Germany)
- Vampir (Technische Universität Dresden, Germany)
- TAU (University of Oregon, USA)

3 Cube Uniform Behavioural Encoding

Cube is a software suitable for displaying a wide variety of performance data for parallel programs (like MPI, OpenMP). It has been designed around a high-level data model of program behaviour called *performance space* [4]. This space is organized in three dimensions.

The *metric dimension* contains all the quantities that can be measured during the runtime of an application, like execution time, cache misses, visits and can also contain *derived* metrics. The *program dimension* contains the program's call tree, which is basically the set of all the calls from the application. Finally, *system dimension* contains the items which are executing in parallel the code. Such items can be processes, or threads, depending on the parallel programming mode. Any point $q(m, c, s)$ can be mapped into a number, which represents the actual measurement for metric m while the process/thread s was executing the call c. This mapping is called the *severity* of the performance space. [5]

Each of the mentioned dimensions are organized into hierarchies. The metric dimension is organized in an inclusion hierarchy where each metric at a lower level is a sub-parent of its parent (see figure 3.1). The program dimension is organized in a call-tree hierarchy, with the same inclusion property. The system dimension is actually the hardware structure of the machine which executed the application and its organized in a multi-level hierarchy consisting on items like processes or threads, up to racks and machine.

Cube has two components: *library* and *display*. The library is able to read and write the data files in the `.cubex` format. The display component can load such files and display information about each dimension with the help of three interactive browsers. The connection between browsers is made in such a way that the user can view one dimension with respect to another dimension (see figure 3.1). For more information about Cube, see the user manual [4] or references [5], [3].

3.1 Using the Cube display

All three browsers represent the performance space. By default, the dimensions are organized form left to right as follows: metric tree, program tree and system tree. Each node from the metric tree represents a metric. The nodes form the program dimension represent all the call paths which form the program tree. System tree contains nodes which represent the machine and all the hardware structure. Each node has an associated value: severity. User can expand or collapse nodes form each tree. By collapsing or expanding a node, its severity changes accordingly (inclusive value or exclusive value; see figure 3.2). When a node is collapsed/expanded it causes the change of the current values in the trees on its right-hand side.

3.2 SystemBoxPlot plug-in

The performance report of an application contains numerical information about the entire runtime of the program, including the severity of all the threads which were executing it. In most cases the number of threads is large, reaching even the order of $n_t = 10^6$. Analysing this amount of data is almost impossible without some summarized schemes which describe numerical behaviour of the data set. One useful scheme which can help the user is a *Box Plot*.

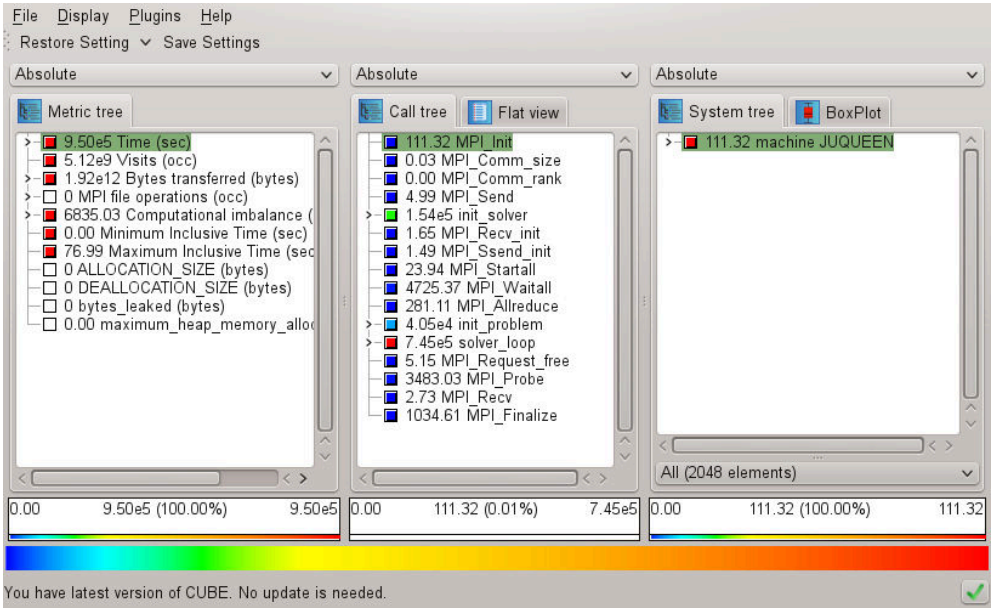


Figure 3.1: Cube-GUI display window.

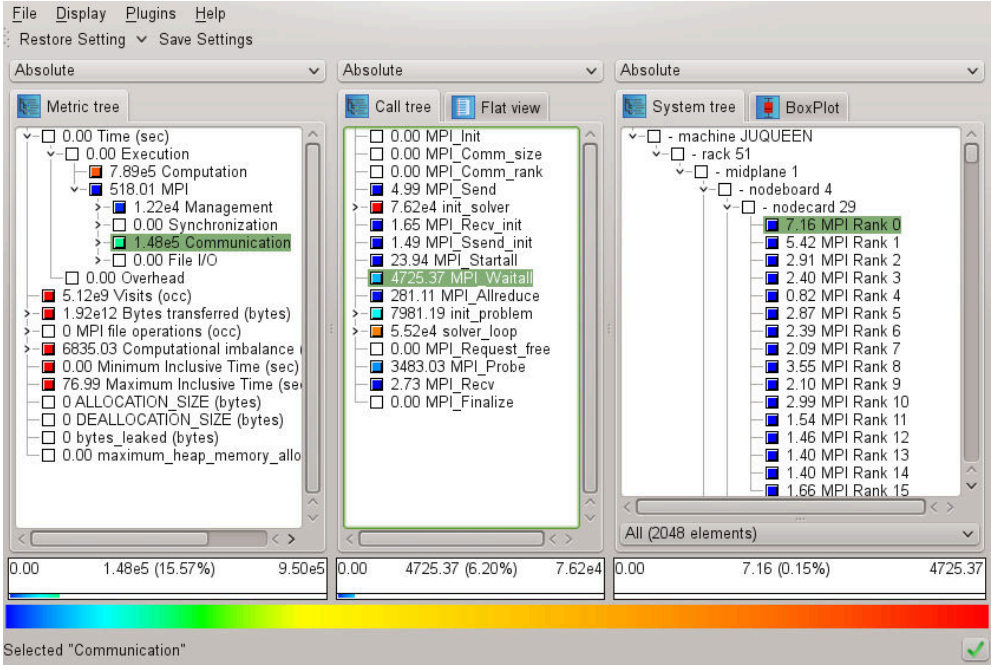
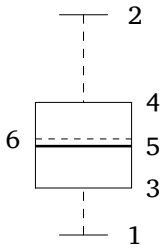


Figure 3.2: Cube display window with expanded metric MPI time and system tree.

3.2.1 Box Plot

In descriptive statistics, a boxplot is a non-parametric way of showing information about numerical data. Typically, a boxplot shows:

1. Minimum and Maximum (1, 2)
2. Q1 and Q3 (3, 4)
3. Mean (5)
4. Median (6)



For such a plot, Cube has already the SystemBoxPlot tool. This plug-in is making a boxplot from the all the severities of the processes/threads in the system tree.

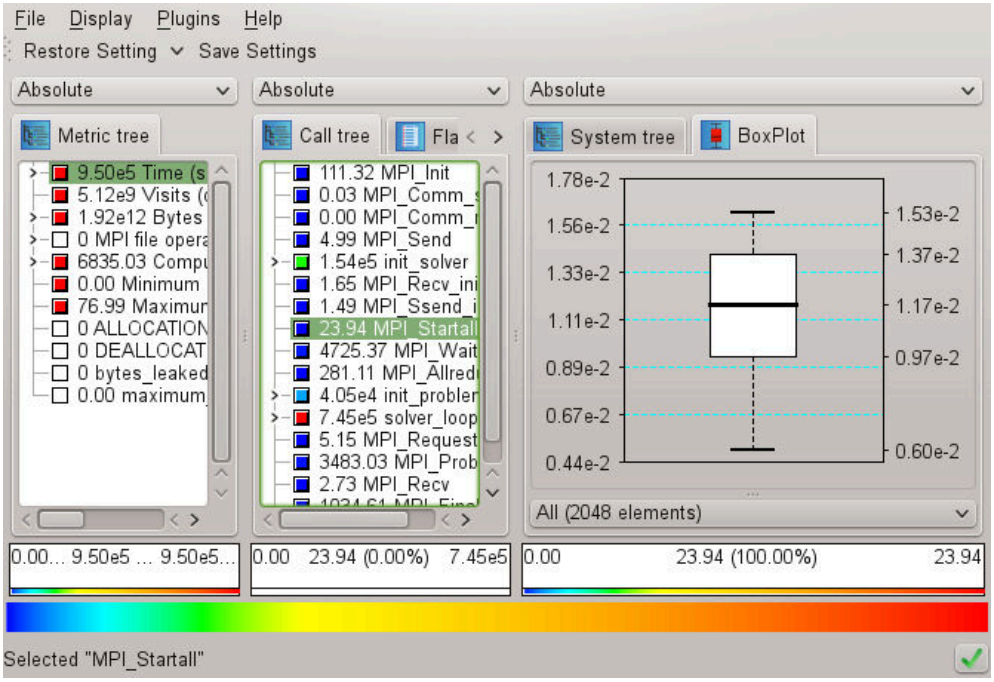


Figure 3.3: The graphical interface of BoxPlot plug-in in Cube display.

Implementing a boxplot in Cube is not complicated due to the fact that for showing only six data points, the algorithm is straightforward. It is advantageous because it shows four main features about the severities: center, spread, asymmetry and outliers.

The only drawback is that a boxplot shows six numbers, and only a *hint* about the distribution of the data. If one is interested into the actual distribution of the data, a boxplot is not the best approach.

Motivation: Cube-GUI must be extended with a new plug-in capable of showing the correct data distribution across the severities from system dimension.

4 Violin Plot – Theoretical Background

In descriptive statistics, violin plot is a method for plotting numerical data. It is a boxplot with a rotated *Kernel density estimation*.

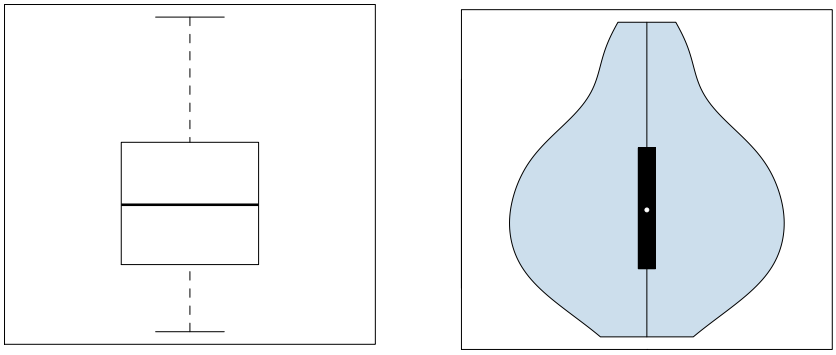


Figure 4.1: The difference between a boxplot and a violin plot.

With the addition of the kernel density estimation to the boxplot, violin plots provides a better information about the shape of the distribution. It can show clusters or bumps in the data keeping in the same time the information provided by the boxplot (see figure 4.1).

4.1 Kernel Density Estimation (KDE)

In statistics, KDE is a way of showing the probability density function of a random variable. Let

x_1	x_2	...	x_n

be an independent sample of numerical data. Then, its kernel density estimator is:

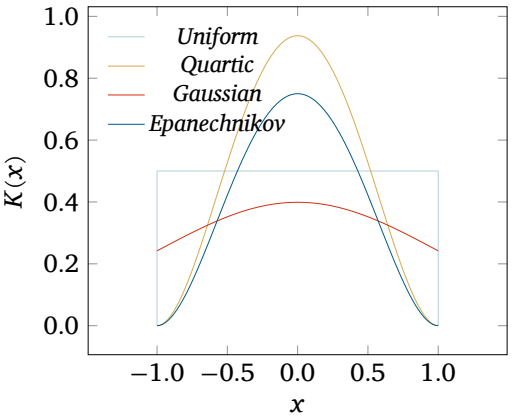
$$KDE(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right) , \tag{4.1}$$

- h – is a smoothing parameter called *bandwidth*, with $h > 0$;
- $K(\cdot)$ is the *kernel*: a non-negative function with mean zero and which integrates to one.

Observation: Choosing the right bandwidth is crucial. An improper value of h can result in *undersmoothing* or even *oversmoothing* of the estimator.

4.1.1 Kernel

In non-parametric statistics, a *kernel* is a weighting function used for estimation techniques. There are several types of kernel functions which can be used in construction of the KDE: *Uniform, Triangle, Epanechnikov, Quartic, Tricube, Triweight, Gaussian, Cosine, Logistic, Sigmoid, Silverman*. In the figure 4.2 are some examples.



Uniform $\frac{1}{2}\mathbf{1}_{|x|\leq 1}$

Quartic $\frac{15}{16}(1-x^2)^2\mathbf{1}_{|x|\leq 1}$

Gaussian $\frac{1}{\sqrt{2\pi}}e^{-\frac{1}{2}x^2}\mathbf{1}_{|x|\leq 1}$

Epanechnikov $\frac{3}{4}(1-x^2)\mathbf{1}_{|x|\leq 1}$

Figure 4.2: Example of kernels.

4.1.2 Influence of bandwidth in estimation

The bandwidth is usually specified in percentages of data range. According to [1], values near 15 % of the data range give good results. Values bigger than 40 % of the range will result in oversmooth of the density and values smaller than 10 % of the range will result in undersmooth (producing strange artefacts). For a good estimation, the data should have at least 30 samples. Importance of h is shown in figure 4.3.

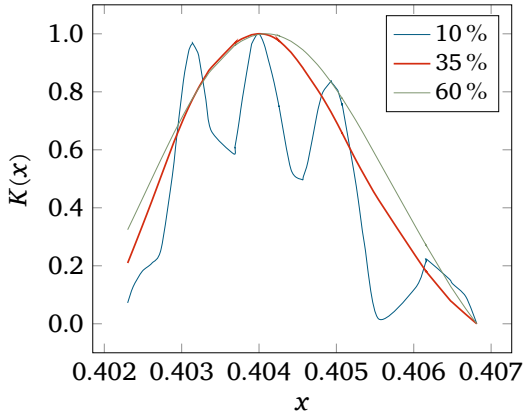


Figure 4.3: Comparison between different values of h for a sample.

Aim: Having now mathematical knowledge about the boxplot and the method for constructing the violin plot, it is possible to implement an algorithm into Cube.

5 Implementation of Violin Plot in Cube

5.1 Workflow

The plug-in will consist from two major parts. The first one takes the severity of all the threads in the system tree as input and computes the KDE and stores it. The second part displays on the screen the KDE using Qt libraries. The workflow can be represented as in figure 5.1.

Figure 5.1: General scheme of the workflow.

5.2 Requirements

Cube is used by the HPC community for a long time so providing users with a new plug-in should be done with care. The user interface must correspond to Cube standards. Having large summary files loaded into Cube, can become a problem because the input data will contain up to millions of entries.

As a result, the algorithm must be fast enough to paint the plot on the screen in a reasonable time; this way assuring a fluent user experience. The entire code of the program must be written in C++ and Qt libraries must be used for displaying the plot.

5.3 First implementation - straightforward algorithm

For this approach, the formula 4.1 is used as a starting point. Direct implementation for computing the KDE can be seen in the following sketch:

Listing 13.1: Naive algorithm for computing KDE

```
// h is the bandwidth
// chosen_kernel is one of the 11 available types of kernels
double sum = 0;
for ( unsigned i = 0; i < n; ++i )
{
     for ( unsigned j = 0; j < n; ++j )
     {
         double temp = ( data[i] - data[j] ) / h;
         sum += chosen_kernel ( temp );
     }
     KDE[i]=sum / ( n * h );
     sum = 0;
}
```

Listing 13.2: Example of subroutine which calculates the kernel

```
// this subroutine computes the Triweight kernel
double chosen_kernel( double arg )
{
    val = abs( arg );
    if ( val > 1 )
    {
        return 0;
    }
```

```
9       double kernel=( 35.0 / 32.0 ) * pow(1 - val * val, 3);
10      return kernel;
11  }
```

The nested loop in the naive violin plot algorithm causes the runtime to be quadratic in n. This corresponds to the measurement shown in figure 5.2.

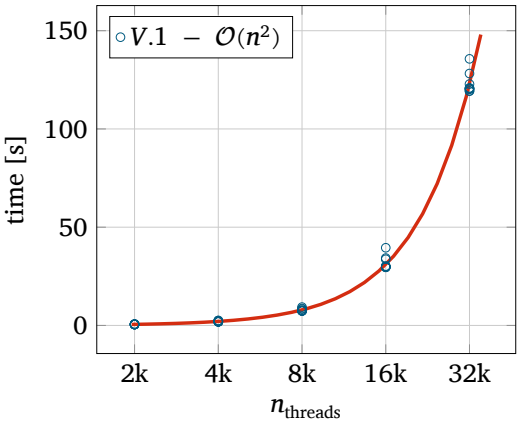


Figure 5.2: Performance measurements for the algorithm.

With this version, the performance is poor even for small files ($n_{\text{threads}} = 32\,000$). The user must wait a long period of time for the violin plot to show up on the display. One important aspect of the program is the process of painting; the algorithm is computing the KDE for each element in the data, then in the Cube window Qt is drawing pixel by pixel each data point. Usually a monitor has 1080 pixels across the vertical axis and painting 32 000 points on a 1080 pixels line will produce *oversampling*. To avoid oversampling a new method of the algorithm should be developed.

5.4 Second version

The key thing for improvement is to reduce data batch for which the KDE will be showed on the screen. If the height of the Cube window is M (pixels), then the KDE should be computed only for M points in the data. As a reminder, most of the time M will be smaller than 1080 pixels.

By *sampling*, a new *data set* can be created where all the points for which the KDE should be calculated are stored. The set will depend on the number of pixels M, which was previously defined and it will contain values normalized with respect to the window size (see figure 5.3). Having the new array created, all it remains is to calculate the KDE for each element. Because in the formula 4.1, $x - x_i$ represents the difference of all the values in the data with respect to x, the same approach should be adopted here. By running from 0 to M (or from `maxp` to `minp`) each element within the original data should be subtracted from the current element of the new data. The element x_i from new data has its *closest* element (as value) somewhere in the original data and that position is unknown; for each new element there will be such a corresponding position. If the position can be somehow calculated, then starting from that element in the old data,

the second loop will split in two parts: one which runs down to the first element and one which runs up to the last element. Within these branches, calculation of KDE can be done as in the first algorithm. The same result can be obtained without splitting the for loop, but in this approach branching the sum could lead to possible improvements. The input data is sorted from minimum value to maximum value, so all the other sets keep the same ascending structure.

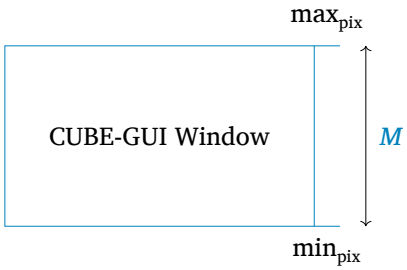


Figure 5.3: The coordinates of the upper and lower part of the Cube window are integer numbers ranging from $\min_{pix}$ to $\max_{pix}$. Once the program gets these two numbers, M will be automatically calculated: $M = \min_{pix} - \max_{pix}$. Because pixel increment is done from top to bottom, the upper part should be subtracted from the lower part.

LISTING 13.3: PERFORMING SAMPLING & CREATING ARRAY WITH NEW DATA AND POSITIONS

```
// maxd and mind are the extreme values from the original data
// minp and maxp are the pixel coordinates of the CUBE window
for ( int i = minp; i >= maxp; --i )
{
  // new element is created and normalized
  double tx = ( minp - i ) / ( minp - maxp ) * ( maxd - mind ) + mind;
  // value is stored in the new data array
  newdata.push_back( tx );
  // position of the closest element from the old data is calulated
  pos = std::lower_bound( data.begin(), data.end(), tx );
  // position is stored in a separate array
  newindex.push_back( pos - data.begin() );
}
```

LISTING 13.4: ALGORITHM FOR THE SECOND METHOD

```
// newdata is the sample
// data is the original array with n entries
int M = minp - maxp;
double sum = 0;
for ( unsigned i = 0; i < M; ++i )
{
    jpos = newindex.at( i ); // setting the starting point of the splitting to the correct index
    // performing the branched summation trough the entire old data
    for ( int j = jpos; j < n; ++j )
    {
        double temp = ( newdata.at( i ) - data.at( j ) ) / h;
        sum += chosen_kernel( temp );
    }
    for ( int j = jpos - 1; j >= 0; --j )
    {
        double temp = ( newdata.at( i ) - data.at( j ) ) / h;
        sum += chosen_kernel( temp );
    }
```

```
19      sum = sum / ( n * h );
20      KDE.push_back( sum );
21      sum = 0;
22      }
23  }
```

M is constant so it will not affect the performance behaviour of the algorithm. As a result, the first loop has been reduced from a linear $\mathcal{O}(n)$ to a constant $\mathcal{O}(M)$, with $M \ll n$. Inside the sampling procedure, the function `lower_bound` has $\mathcal{O}(\log_2 n)$ complexity while the branched summation still behaves linearly $\mathcal{O}(n)$. By making calculation of the total complexity of the algorithm, the following result is expected:

$$R \;=\; \mathcal{O}(M \log_2 n) + \mathcal{O}(Mn) \;.$$

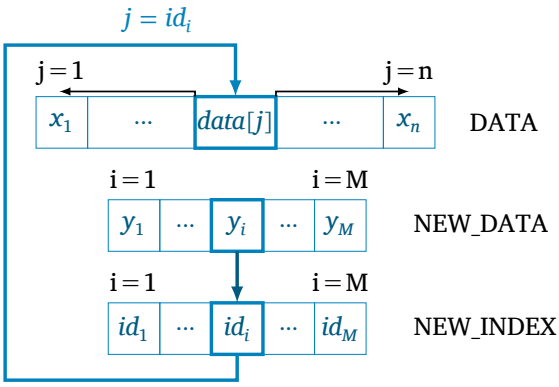


Figure 5.4: The new data will have a corresponding new index pointing to the position of its closest element in the original data.

The obtained results verify the expected complexity of the program and one can observe that the performance has improved by a large factor even for a file with data from 32 000 threads and this result was achieved only by using a sampling procedure. But still, good performance should be obtained for $n = 10^6$, so further optimization in this algorithm must be made. The major impact is represented by the linear part of the calculation and reducing the $\mathcal{O}(n)$ to a smaller complexity should improve the execution time of the plug-in even more.

5.5 Third version

In order to decrease the branched loop from n to a smaller value, the algorithm must be adjusted in such a way that will behave selectively. In the kernel subroutine, the condition for which the function will return zero is

$$\gamma \;=\; \left[\texttt{(newdata.at(i) - data.at(j)) / } h\right] \notin [-1,1] \;,$$

and the summation term will not change after this condition due to the zero result of $K(\gamma)$. If it would be possible to know after how many iterations γ will be outside the

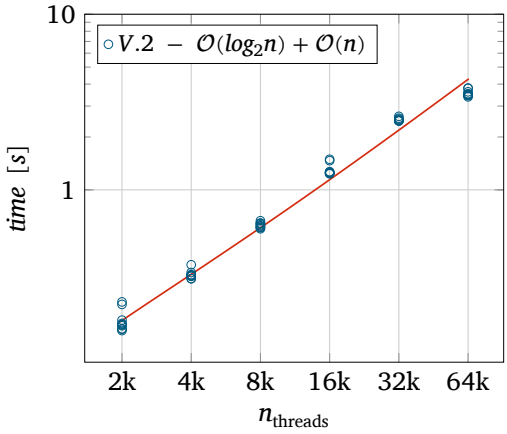


Figure 5.5: Performance measurements for the algorithm in a loglog plot.

range, then the branched loop can stop (with a halt condition) before completing the entire $1 \longleftrightarrow n$ loop (see figure 5.4). As a result, a cutting limit for each branch must be found so that the algorithm will finish computing the KDE before the entire original data is completed, and the linear $O(n)$ behaviour will become $\mathcal{O}(\zeta)$, where ζ is smaller than n. The difference of the approach is shown in figure 5.6.

The importance of ζ lies in the fact that one can find the exact location in the data where

$$newdata[i] \; - \; data[j] < h, \; j \in \left[\; j_{\text{left\_limit}} \; , \; j_{\text{right\_limit}} \; \right] \; .$$

With the new limits, it is possible to go back in the second version and introduce them as cutting conditions for the branched loop. Unfortunately by doing this, the improvement in time performance will be insignificant (due to linear complexity of the loop and `find_if`). A better way is to make an approximation by calculating only the first kernel of

$$\eta \; = \; \frac{newdata[i] - data\left[\; newindex[i] \; \right]}{h} \; ,$$

and then just multiply this result (call it with the allowed number if iterations, which is ζ. Even if this can result in some inaccuracy for KDE, it will get rid off the branched loop, which could help improving the execution time even more, because the algorithm will compute M KDEs trough a simple multiplication $KDE \; = \; \eta\zeta$. Each element in the `newdata` will have its corresponding `newindex` for the closest element in the original data as well as the left and right limits. The entire problem can be solved now in only one for loop which runs through the number of pixels.

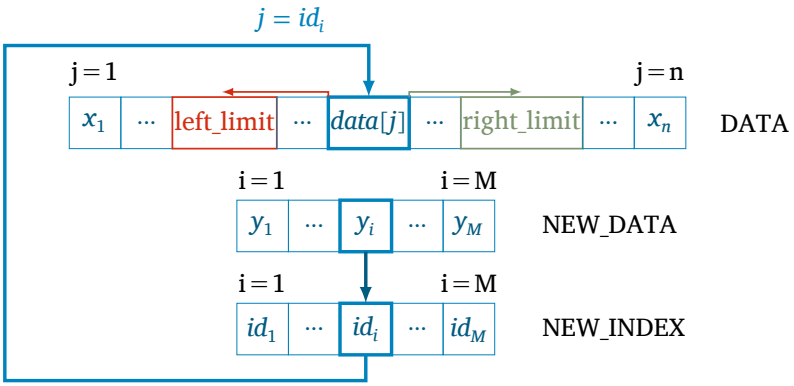


Figure 5.6: Improved selective algorithm.

LISTING 13.5: SELECTIVE ALGORITHM OF THE THIRD VERSION

```
// find left and right limits for data
for ( unsigned i = 0; i < M; ++i )
{
     tx = ( minp - i ) / ( minp - maxp ) * ( maxd - mind ) + mind;
     newdata.push_back( tx );
     // find (in original data) the position of closest element for newdata
     glob_pos = std::lower_bound( data_batch.begin(), data_batch.end(), tx );
     // store the position
     newindex.push_back( glob_pos - data_batch.begin() );
     // find the left limit
     pos = std::find_if( data_batch.begin(), glob_pos, k_condition_upper );
     // store position of left limit for each element in newdata
     iterations_left.push_back( std::distance( pos, glob_pos ) );
     // find the right limit
     pos = std::find_if( glob_pos, data_batch.end(), k_condition_lower );
     // store position of the right limit for each element in newdata
     iterations_right.push_back( std::distance( glob_pos, pos ) );
}
// calculate kernel
for ( unsigned i = 0; i < M; ++i )
{
     jpos = newindex.at( i );
     left_limit  = iterations_left.at( i );
     right_limit = iterations_right.at( i );
     zeta = right_limit - left_limit;
     double rez = ( newdata.at( i ) - data_batch.at( jpos ) ) / width;
     sum = zeta * chosen_kernel( rez ); // improved calculation(approximation)
     sum = sum / ( n * h );
     KDE.push_back( sum );
}
```

Finding these limits can be done with a C++ template called `find_if`. This function searches and returns the first position in an array where an arbitrary condition is true. For the left limit the required condition is that `newdata[i] − data[j]` $< h$ and for the right limit the condition is `newdata[i] − data[j]` $> h$. The reason is that beyond this limits, the condition for the value inside the kernel to be in the range of $[-1, 1]$ is not any more valid. The complexity of `find_if` is up to linear between first and last element.

5.5.1 A CLOSER LOOK UPON h

During the description of all the algorithms, h was ignored. It is necessary to check if the value of the bandwidth has some influence on the behaviour of the algorithm. Importance of h for the KDE was already mentioned, so analysing the change in performance of the program with respect to the bandwidth is crucial now. ζ is the number which shows the location of all the elements in the original data which satisfy the condition for `chosen_kernel` subroutine. As a reminder, that condition is

$$\left|\frac{newdata[i] - data[j]}{h}\right| < 1 \ .$$

This formula clearly shows the importance of bandwidth: a larger value of h will result in a larger interval between the left limit and the right limit, which means that the value of ζ is closely related to the value of h :

$$\zeta \ = \ f(h) \ .$$

The plug-in allows the user to adjust the bandwidth in a range from 0 % to 100 % (see 6.1). During the measurements, the change of h did not produce significant changes in performance. However, for a large array, it is possible that h will have an influence on execution time due to the larger interval that `find_if` needs until the halt condition is satisfied.

Going back to the performance of the program, the expected behaviour depends on the complexity of this algorithm. Based on the properties of the third version total complexity can be calculated with the following formula: $R \ = \ \mathcal{O}(M \log_2 n) \ + \ \mathcal{O}\left(M\zeta(h)\right) \ + \ \mathcal{O}(M)$ where the first term belongs to `find_if` function, the second term corresponds to calculation of the limits and the third term is from the actual computation of KDE in the program. Because M is constant during program execution, one can neglect the influence of third term, resulting in a final complexity of

$$\mathcal{O}\left(M \log_2 n\right) \ + \ \mathcal{O}\left(M\zeta(h)\right) \ .$$

As it was intended from the beginning, the linear term in n was reduced to a smaller value, which could produce more improvements in time performance.

5.5.2 FINAL RESULTS

The performance measurements for all three versions can be seen in the figure below:

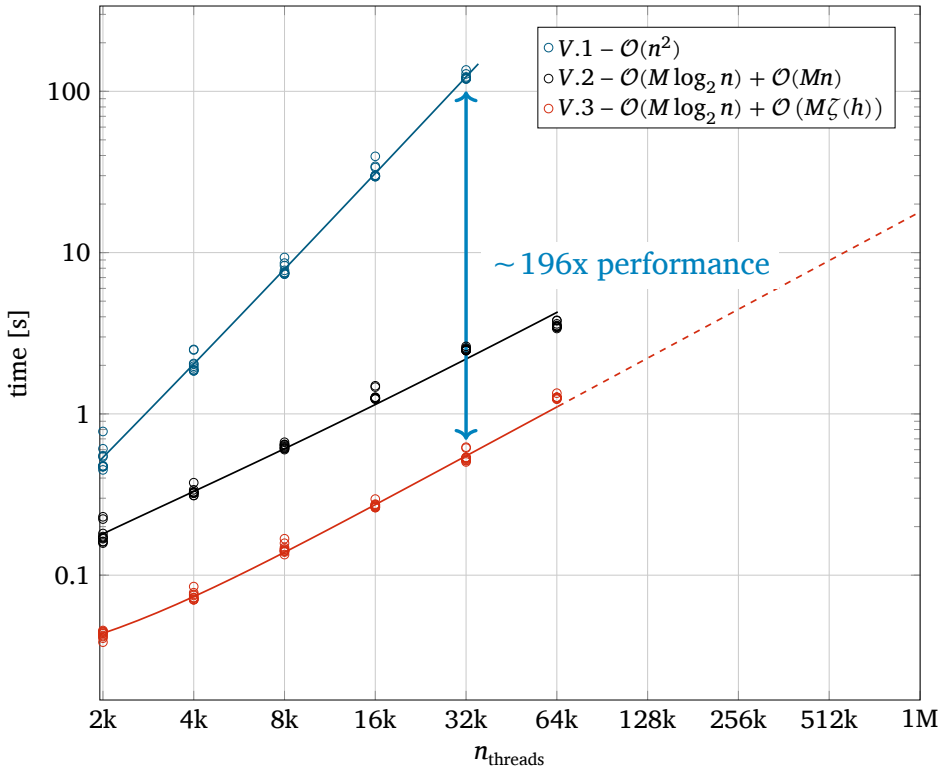


Figure 5.7: Performance measurements for all the versions in a loglog plot.

6 The user interface of the plug-in

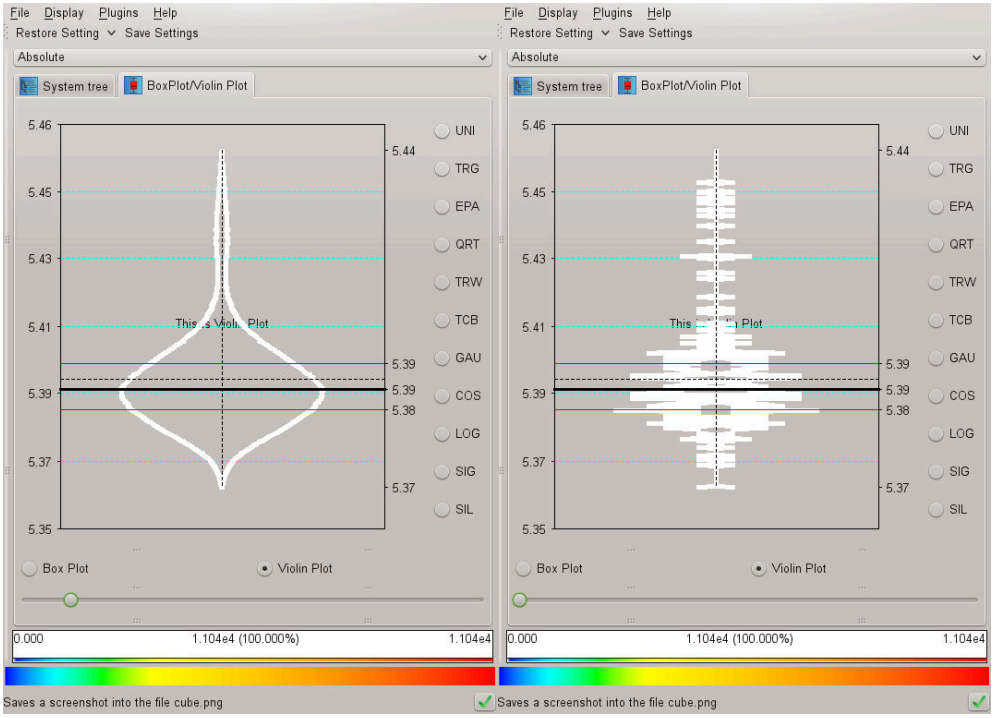


Figure 6.1: Graphical representation of violin plot in Cube. Information given by boxplot can still be visualized. Users can choose the kernel (right radio buttons) and also change the value of the bandwidth (bottom slider). Different value of h will result in undersmooth or oversmooth. In this example the undersmooth problem can be seen in the right picture.

7 Summary & Conclusions

Having a violin plot for the Cube software will provide the distribution of data across the system tree which can potentially help the HPC application. A closer look at the system threads severity, especially the groups with high values, will reveal potential *hardware malfunction* (e.g. communication problems between nodes, processors which are thermal throttling). As a result, the violin plot will not only improve the HPC application, but also the HPC system.

The third version of the plug-in has the best performance of all the versions, and it is reasonably fast when working with summary reports of medium size. Based on extrapolation a calculation time of $t \approx 20\,\mathrm{s}$ for a violin plot on a system with 10^6 threads.

The plug-in provides the user possibility of choosing the kernel estimation and value of h in order to give a *live feedback* with the change in shape of the violin plot (see figure 6.1).

Observation: The choice of kernel should not make any major change in the shape of the KDE and the kernel should not influence the time execution of the program (even if the function require different number of mathematical operations). Fortunately, the kernel type is not important in time performance.

Performance measurements for the algorithms were realized with SCORE-P, using *sampling*. Sampling is an interrupt-based event measurement technique. For more details on performance measurements with SCORE-P see reference [2]. Below, a graphical representation of Violin Plot plug-in working mechanism is shown.

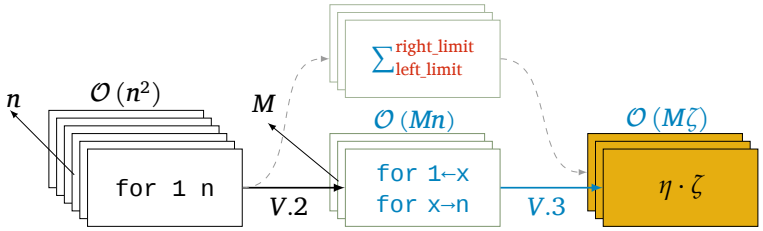


Figure 7.1: This figure contains a short review for all the version of the algorithm. Each element in a stack represents an iteration for the calculation of KDE. First version (represented by n elements in the left stack, where each element is a KDE) was the naive implementation directly from equation 4.1, which lead to quadratic behaviour. By sampling, the complexity was reduced in the second version (represented by the middle stack, with only M elements) to $\mathcal{O}(Mn)$, keeping the branched loop linear in n. The alternative of $V.2$ is to introduce a halt condition in the branched loop with the help of cutting limits, but the time performance will not change. Finally, the third version (represented by the right stack with M elements, where each element is just a multiplication of two numbers) removes the branched loop by approximating the KDE with $KDE = \eta\zeta$, reducing in this way the runtime to $\mathcal{O}(M\zeta)$.

8 Outlook / Future work

Because most of the time, HPC applications are running on machines with allocated space on the order of $n_{\text{threads}} = 10^4 - 10^5$, the V.3 of the plug-in is ready for a possible implementation into a new version release of CUBE. Even though, for a large file 20 s can be a drawback for the user, the third version has possibilities for further improvements: reduction of ζ with a new selective algorithm or some UI tweaks, so there is a possibility of reducing the waiting time even more.

References

[1] J. L. Hintze and R. D. Nelson. **Violin plots: a box plot-density trace synergism**. *The American Statistician*, 52(2):181–184, 1998.

[2] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony, et al. **Score-p: A joint performance measurement run-time infrastructure for periscope, scalasca, tau, and vampir**. In *Tools for High Performance Computing 2011*, pages 79–91. Springer, 2012.

[3] P. Saviankou, M. Knobloch, A. Visser, and B. Mohr. **Cube v4: From performance report explorer to performance analysis tool**. *Procedia Computer Science*, 51:1343–1352, 2015.

[4] F. Song and F. Wolf. **Cube user manual**. *University of Tennessee, Innovative Computing Laboratory*, 2004.

[5] I. Zhukov, C. Feld, M. Geimer, M. Knobloch, B. Mohr, and P. Saviankou. **Scalasca v2: Back to the future**. In *Tools for High Performance Computing 2014*, pages 1–24. Springer, 2015.