Parallelization of a Differential Algebra Framework

Federal Institute of Technology ETH Zurich

Bachelor Thesis

in Computational Science and Engineering CSE

Department of Mathematics

written by

Matthias Frey

supervised by

Dr. A. Adelmann

August 2013

Abstract

The purpose of this thesis is the partially parallelization of a differential algebra library with OpenMP. Through execution time analysis with a reference program, costly parts got located and more deeply considered for possible optimizations and parallelization.
A roofline model shows that the considered function is memory bound wherefore peak performance can't be achieved on the applied CPU.
The limitation of OpenMP with respect to the memory control which is important on NUMA architectures, is a problem. Therefore, manual threading or corresponding compiler flags could be the answer for better performance. A comparison between g++ 4.7.2 and the Intel compiler icc 13.0.1 serves as verification of the results.

Contents

1 Introduction **8**
- 1.1 Motivation . . . 8
- 1.2 Scope of Work . . . 8
- 1.3 Differential Algebra . . . 9
  - 1.3.1 Particle Motion . . . 9
  - 1.3.2 Differential Algebra in ${}_1D_1$ . . . 9
  - 1.3.3 Differential Algebra in ${}_nD_\nu$ . . . 10
- 1.4 OpenMP . . . 11
- 1.5 SBEND Program . . . 12

2 Parallelization **13**
- 2.1 Analysis . . . 13
  - 2.1.1 FVps<T,N>::ExpMap . . . 14
  - 2.1.2 FTps<T,N>::multiply . . . 15
  - 2.1.3 FTps<T,N>::operator+ . . . 16
  - 2.1.4 FTps<T,N>::derivative . . . 17
- 2.2 Methods . . . 18
  - 2.2.1 FTps<T,N>::multiply . . . 18
  - 2.2.2 FTps<T,N>::operator+ . . . 18
  - 2.2.3 FTps<T,N>::derivative . . . 19
  - 2.2.4 Non-Uniform Memory Access (NUMA) . . . 20

3 Results **21**
- 3.1 Cluster Architecture . . . 21
  - 3.1.1 Merlin4 . . . 21
  - 3.1.2 Peak Performance . . . 21
- 3.2 Measurements . . . 22
- 3.3 Scalings . . . 23
  - 3.3.1 FTps<T,N>::multiply . . . 23
  - 3.3.2 FTps<T,N>::operator+ . . . 28
  - 3.3.3 FTps<T,N>::derivative . . . 32
- 3.4 Discussion . . . 33
- 3.5 Conclusion . . . 34
- 3.6 Acknowledgement . . . 34

4 Appendix **35**
4.1 FTps<T,N>::multiply 35
4.1.1 Versions 35
4.1.2 Timings 38
4.2 FTps<T,N>::operator+ 41
4.2.1 Versions 41
4.2.2 Timings 43
4.3 FTps<T,N>::derivative 45
4.3.1 Versions 45
4.3.2 Timings 46

List of Figures

1.1 Dimension of ${}_nD_\nu$ 11
1.2 Fork-join model 11

2.1 CPU time usage 13
2.2 Maximum loop size of FTps<T,N>::multiply 15
2.3 Execution ratio 16
2.4 Maximum loop size of FTps<T,N>::operator+ 16
2.5 Data access in FTps<T,N>::derivative 17
2.6 Maximum loop size of FTps<T,N>::derivative 18

3.1 Roofline model of CPU E5-2670 and X5670 22
3.2 Speedup version M7, M13 and M15 23
3.3 Speedup versions M1 to M6 24
3.4 Speedup versions M8 to M11 24
3.5 Speedup versions M12, M14 and M16 24
3.6 NUMA vs. non-NUMA of FTps<T,N>::multiply 25
3.7 Speedup of FTps<T,N>::multiply for various orders 26
3.8 Improved speedup of FTps<T,N>::multiply for various orders . . 26
3.9 ICC vs. GCC for FTps<T,N>::multiply 27
3.10 Roofline model of FTps<T,N>::multiply 28
3.11 Speedup of FTps<T,N>::operator+ 28
3.12 Speedup of O2 with different scheduling strategies 29
3.13 Speedup of FTps<T,N>operator+ version O2d 29
3.14 NUMA vs. non-NUMA of FTps<T,N>::operator+ 30
3.15 Speedup of FTps<T,N>::operator+ for various orders 30
3.16 Improved speedup of FTps<T,N>::operator+ for various orders . 31
3.17 ICC vs. GCC for FTps<T,N>::operator+ 31
3.18 Inverse speedup of FTps<T,N>::derivative 32
3.19 NUMA vs. non-NUMA of FTps<T,N>::derivative 32

List of Tables

3.1 Node characteristics of Merlin4. [11] 21

4.1 Timings of FTps<T,N>::multiply (part 1) 38
4.2 Timings of FTps<T,N>::multiply (part 2) 38
4.3 Timings of FTps<T,N>::multiply (part 3) 39
4.4 Timings of FTps<T,N>::multiply for various orders 39
4.5 Timings of FTps<T,N>::multiply for various orders with if-clause 39
4.6 Serial timings of FTps<T,N>::multiply for various orders 40
4.7 Timings of FTps<T,N>::multiply GCC vs. ICC 40
4.8 Timings of FTps<T,N>::operator+ (part 1) 43
4.9 Timings of FTps<T,N>::operator+ (part 2) 43
4.10 Timings of FTps<T,N>::operator+ for various orders 44
4.11 Timings of FTps<T,N>::operator+ for various orders with if-clause 44
4.12 Serial timings of FTps<T,N>::operator+ for various orders . . . 44
4.13 Timings of FTps<T,N>::operator+ GCC vs. ICC 45
4.14 Timings of FTps<T,N>::derivative 46

Listings

1.1 Hamiltonian construction in *sbend* 12

2.1 Exception handling before 14
2.2 Exception handling afterwards 14
2.3 Structure of FTps<T,N>::multiply 15
2.4 While-loop 17
2.5 For-loop 17
2.6 Structure with sections 19
2.7 Structure with one parallel block 19
2.8 Structure without parallel block 19
2.9 Strategy for FTps<T,N>::derivative. 19

3.1 Code snippet of version M10. 25

4.1 Version M1 and M2 35
4.2 Version M3 35
4.3 Version M4 36
4.4 Version M5 and M6 36
4.5 Version M7 36
4.6 Version M8 37
4.7 Version M9 and M11 37
4.8 Version M10 37
4.9 Version M12, M13 and M16. 37
4.10 Version M14 and M15. 38
4.11 Version O1 41
4.12 Version O2 41
4.13 Version O3 42
4.14 Version O2a, O2b and O2e. 42
4.15 Version O2c and O2d 42
4.16 Version D1 45
4.17 Version D2 and D3 45
4.18 Version D4 and D5 45

Chapter 1

Introduction

1.1 Motivation

The Differential Algebra Framework (short DA) is part of OPAL, an **O**bject-Oriented **P**arallel **A**ccelerator **L**ibrary of the AMAS (**A**ccelerator **M**odelling and **A**dvanced **S**imulations [15]) Group of PSI (**P**aul **S**cherrer **I**nstitut), which is used for charged particle optics [1].
Based on the mathematical background of Taylor series expansions a DA program contains a lot of loops. These loops can be very time consuming depending on the truncation order and the problem size. Despite lots of code optimizations concerning these loops, parallelization is an unavoidable method for making code running faster.
There are several techniques and libraries available with the purpose of code parallelization but their explanation is out of the scope of this thesis. We restrict ourselves on OpenMP, a library which is able to parallelize every source code through small changes.

1.2 Scope of Work

The purpose of this bachelor thesis is the development of a concept for the parallelization of the Differential Algebra Framework using OpenMP. The provided code example, called *sbend*, serves as reference and is the basis for benchmarks on the partially parallelized library. The Merlin4 Cluster at PSI is provided for scaling measurements.

1.3 Differential Algebra

1.3.1 Particle Motion

Considering the Γ-space with N particles $\vec{\xi}_i = (\vec{q}_i, \vec{p}_i) \in \mathbb{R}^d \times \mathbb{R}^d$ the Hamiltonian for describing the time evolution of the system can be written as

$$\frac{d\vec{q}_i}{dt} = \frac{\partial H_N}{\partial \vec{p}_i} \ , \ \frac{d\vec{p}_i}{dt} = -\frac{\partial H_N}{\partial \vec{q}_i}. \tag{1.1}$$

From (1.1) and the Liouville equation we end up with the Liouville operator L_N. If f defines a distribution function, then the Liouville operator looks as follows:

$$L_N f_N \equiv \frac{\partial f_N}{\partial t} + \{f_N, H_N\}, \tag{1.2}$$

where the Poisson Bracket is denoted by

$$\{f_N, H_N\} = \sum_{i=1}^{N} \left(\frac{\partial f_N}{\partial \vec{q}_i} \cdot \frac{\partial H_N}{\partial \vec{p}_i} - \frac{\partial f_N}{\partial \vec{p}_i} \cdot \frac{\partial H_N}{\partial \vec{q}_i} \right). \tag{1.3}$$

On closer examination of (1.3) the parts of (1.1) can be seen. [2]
The motion of a particle through an element in s is described by

$$\vec{\xi}(s) = e^{:H:s} \vec{\xi}(0) = \mathcal{M} \circ \vec{\xi}(0). \tag{1.4}$$

The syntax $:H:s$ in the exponential function of (1.4) is equivalent to the Poisson Bracket notation in (1.3). [2]
Using the *Dragt-Finn factorization* (see [5],p.159-164) one can rewrite (1.4) into the form

$$\vec{\xi}(s) = \left(\prod_{i=1}^{N} e^{:H_i:s} \right) \vec{\xi}(0). \tag{1.5}$$

The H_i represent Lie operators of $\mathcal{M}$ whereat each of them is a homogeneous polynomial of degree i. [3]
The factorization gives us a tool to compute the map piecewise by the construction of Taylor series of all H_i. These series are evolved by using the Differential Algebra ${}_nD_\nu$.

1.3.2 Differential Algebra in ${}_1D_1$

Before starting with the general case, we discuss the 1-dimensional DA. If we obtain two pairs (q_0, q_1), (r_0, r_1) with $q_0, q_1, r_0, r_1 \in \mathbb{R}$, then the fundamental operations are defined as follows:

$$(q_0, q_1) + (r_0, r_1) = (q_0 + r_0, q_1 + r_1) \tag{1.6}$$

$$t \cdot (q_0, q_1) = (t \cdot q_0, t \cdot q_1) \ \forall t \in \mathbb{C} \tag{1.7}$$

$$(q_0, q_1) \cdot (r_0, r_1) = (q_0 \cdot r_0, q_0 \cdot r_1 + q_1 \cdot r_0). \tag{1.8}$$

It can easily be checked that the identity element of the algebra is $(1, 0)$ whereas the other way around, we have the infinitesimal element:

$$(0, 0) < d := (0, 1) < (q, 0) = q \ \forall q \in \mathbb{R}. \tag{1.9}$$

If $q_0 \neq 0$ for an element $(q_0, q_1) \in {}_1D_1$, then its multiplicative inverse is defined by

$$(q_0, q_1)^{-1} = \left(\frac{1}{q_0}, -\frac{q_1}{q_0^2}\right). \tag{1.10}$$

The root is computed if and only if $q_0 > 0$ with

$$\sqrt{(q_0, q_1)} = \left(\sqrt{q_0}, \frac{q_1}{2\sqrt{q_0}}\right). \tag{1.11}$$

With the map $\partial : {}_1D_1 \mapsto {}_1D_1$ we introduce the most useful aspect for beam physics:

$$\partial(q_0, q_1) = (0, q_1). \tag{1.12}$$

Let f be a function in ${}_1D_1$ and $q \in \mathbb{R}$, then

$$f(q + d) = (f(q), f'(q)), \tag{1.13}$$

where f' denotes the first derivative of f. The brace notation [.] is short for the evaluation at the origin:

$$[f] = (f(0), f'(0)). \tag{1.14}$$

(see [5], p.86-91)

1.3.3 Differential Algebra in ${}_nD_\nu$

The shorthand notation is comparable to $\mathbb{C}^n(\mathbb{R}^\nu)$ which means the space of n times continuously differentiable functions on vector space $\mathbb{R}^\nu$.
Let $f, g \in {}_nD_\nu$, the arithmetic operations are defined by

$$[f] + [g] = [f + g], \tag{1.15}$$

$$t \cdot [f] = [t \cdot f], \tag{1.16}$$

$$[f] \cdot [g] = [f \cdot g]. \tag{1.17}$$

The map $\partial_k : {}_nD_\nu \mapsto {}_nD_\nu$ for $k = 1, ..., \nu$ is specified as

$$\partial_k f = \left[p_k \cdot \frac{\partial f}{\partial x_k}\right] \tag{1.18}$$

where $p_k(x_1, ..., x_\nu) = x_k$. For all $f, g \in {}_nD_\nu$, the algebra follows the known rules:

$$\partial_k([f] + [g]) = \partial_k([f + g]), \tag{1.19}$$

$$\partial_k([f] \cdot [g]) = \partial_k[f] \cdot [g] + [f] \cdot \partial_k[g], \tag{1.20}$$

The dimension of ${}_nD_\nu$ increases in an exponential manner with the order n by keeping the number of variables ν constant (see Fig. 1.1):

$$\dim {}_nD_\nu = \frac{(n + \nu)!}{n!\nu!}. \tag{1.21}$$

(see [5], p.91-96)

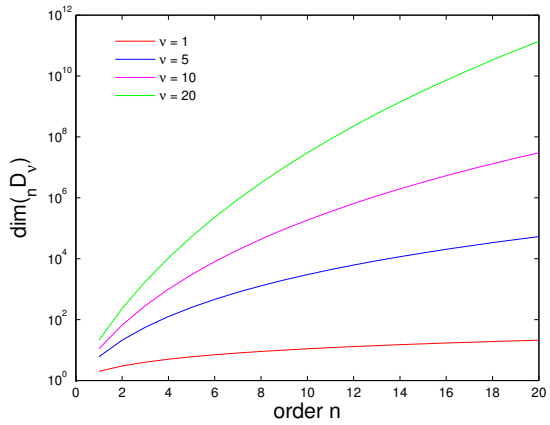


Figure 1.1: Dimension of ${}_nD_\nu$. *Note*: Logarithmic scale in y-direction.

1.4 OpenMP

OpenMP – version 3.0 appeared in August 2008 – is a C/C++ and Fortran library for parallel programming. In C++ the library can easily be included with #include <omp.h> (more details are given in [12]).
OpenMP works with the fork-join model, where threads are generated by the master thread before a parallel region and joined afterwards (see Fig. 1.2). The library is very useful in connection with for-loops, which do not contain any break-statements, but inappropriate for while-loops.

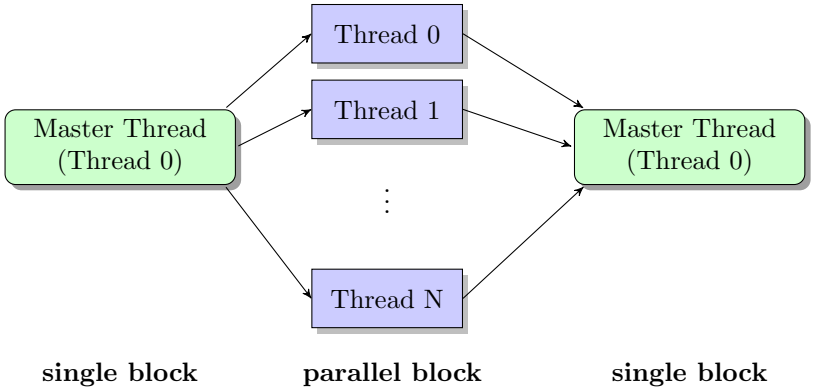


Figure 1.2: Example of the fork-join model.

OpenMP contains many useful functions and directives. It also includes some scheduling strategies where one can control the assignment of chunks of an array to a thread during a loop.

1.5 SBEND Program

The reference program written by Dr. A. Adelmann is called *sbend*. It's used to check if the parallelized version still returns the same results as the serial one and to do scaling measurements.
The program computes the mapping of the particle configuration from the initial state into the final state using (1.5) where the Hamiltonian is given by

$$H = -(1+hx)\left(eA_s + \sqrt{\left(\frac{E}{c}\right)^2 - (mc)^2 - (p_x - eA_x)^2 - (p_y - eA_y)^2}\right) \quad (1.22)$$

with curvature h, particle charge e, particle mass m, velocity of light c, energy E, magnetic vector potential $\vec{A} = (A_x, A_y, A_s)$ and the momentum in x-direction p_x or, respectively, y-direction p_y (see [4], p.2).
After the computation of (1.5) with (1.22) the identity map is substituted and the result is printed.
Listing 1.1 shows the construction of the Hamiltonian and the computation of the result, called myMap.

```
// Construct the Hamiltonian H about the closed orbit.
// (1) Define variables.
double scale = 1.0;
Vector co(0.0);
static const Series px = Series::makeVariable(PX);
static const Series py = Series::makeVariable(PY);
static const Series pz = Series::makeVariable(PZ) + 1.0;
static const Map ident;
Map myMap;
Map translate = co + ident;

// (2) Construct kinematic terms.
double kin = 1.0/(ga*be);
Series E_trans = sqrt((pz*pz+kin*kin).substitute(translate),order)/be;
Series pz_trans = sqrt((pz*pz-px*px-py*py).substitute(translate),order);

// (3) Build vector potential in straight reference frame.
Series As = buildSBendVectorPotential(field,h)*scale;
As.setTruncOrder(Series::EXACT);

// (4) Build Hamiltonian.
Series H_trans = E_trans - pz_trans + As.substitute(translate);
H_trans.setMinOrder(2);

myMap = ExpMap((-length) * H_trans).substitute(myMap);
```

Listing 1.1: Code snippet showing the construction of the Hamiltonian in *sbend*.

Chapter 2

Parallelization

2.1 Analysis

Before developing a parallelization strategy, the *sbend* program was analysed according to its time consuming parts. For this purpose GPROF [14], a tool for profiling a code, was applied.
In order to have better statistics about the library usage concerning the computation of the Taylor series expansions, the original program was modified the following way: The global truncation order was set to 9 – which results in a longer computation time – and the map has been substituted forty times.

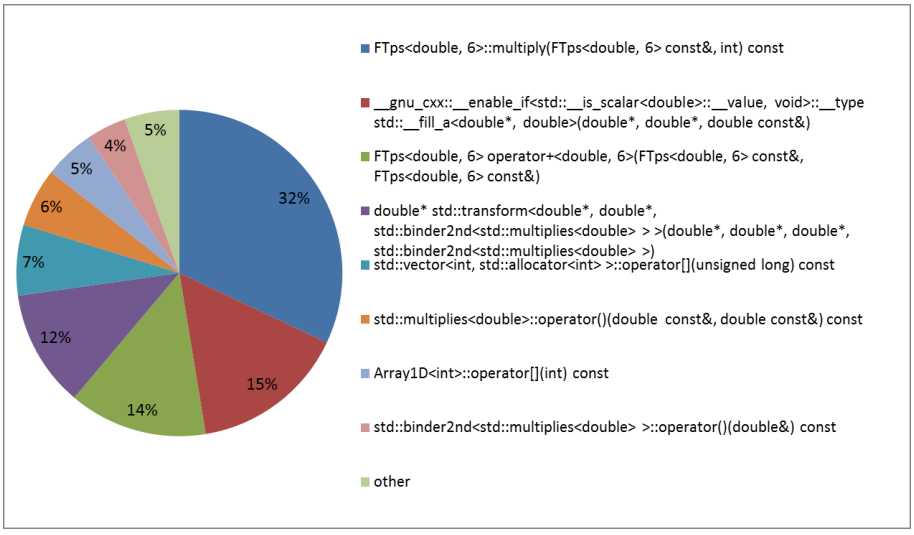


Figure 2.1: CPU time usage of *sbend* with truncation order 9 and 40 map substitutions in vector space $\mathbb{R}^6$.

From Fig. 2.1 one obtains that the program spends one third of its time in FTps<T,N>::multiply.
Each, the STL algorithm std::fill and FTps<T,N>::operator+, use about 15 % of execution time.

The two functions of FTps (fixed truncated power series) are called through FVps<T,N>::ExpMap which is responsible for building the Taylor series for the Hamiltonian.
Due to the result of the runtime analysis the general concept follows the idea of parallelizing the most time consuming parts of each function obtained from Fig. 2.1. These considerations are treated in the next few subsections where the main improvements and the parallelization plans are explained.

2.1.1 FVps<T,N>::ExpMap

The *sbend* program calls firstly the ExpMap function which is part of the FVps (fixed vector truncated power series) template class. It's used to compute the maps for the particle motion which was described in section 1.3 in detail.
Although this function contains nested loops, it's not useful to parallelize them, because their dimension belongs to the dimension of the particles which is in general chosen to be six due to the vector space $\mathbb{R}^6$ with (x, y, s, p_x, p_y, p_s). Inside these loops there are some function calls to FTps<T,N>::multiply.
A first improvement could already be done by passing the exception check to the end of the function and replacing the if-statement inside the loop by an assignment to a variable of type bool (Listing. 2.1 and 2.2). A measurement yielded that one execution of this operation is about twice as fast as the if-statement. However, pipelining with good branch predictions in loops allows the if-statement to be almost equally fast.

```
for(/*increment*/) {
 if(k>MAX_ITER) {
  //throw an exception
 }
 //do something
}
```

Listing 2.1: Exception handling before (slow version).

```
bool exception = false;
for(/*increment*/) {
 exception = (k>MAX_ITER);
 //do something
}
if(exception) {
 //throw an exception
}
```

Listing 2.2: Exception handling afterwards (fast version).

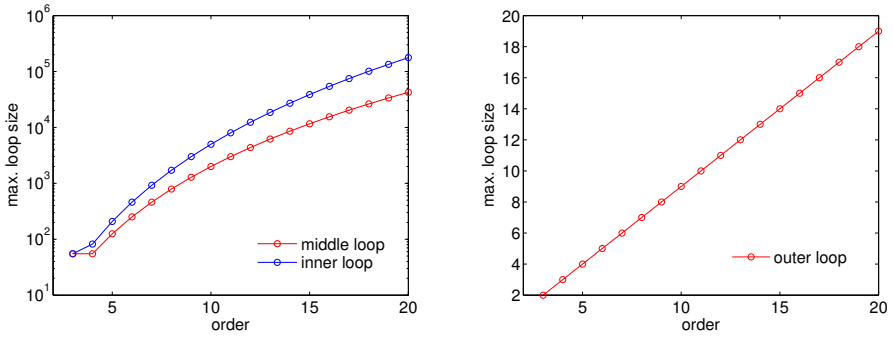
2.1.2 FTps<T,N>::multiply

The analysis of the *sbend* program has shown that it spends about a third of its time in the multiply function of the FTps template class. The function consists of three nested loops (see Listing 2.3) where in the innermost loop the operation $\vec{y}_i = \vec{z}_i \cdot \vec{x}_i + \vec{y}_i$ on a whole array − describing a Taylor series expansion − is executed.

```
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 // set ending indices
 last_f = orderEnd(std::min(f_max, maxOrder - gOrd));
 last_g = orderEnd(gOrd);
 // do multiplies for each entry (of order gOrd) in rhs
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   const Array1D<int> &prod = FTpsData<N>::getProductArray(j);
   for(int i = first_f; i < last_f; i++) {
    result[prod[i]] += f[i]*g[j];
   }
  }
 }
 first_g = last_g; //reset starting index for next order of rhs
}
```

Listing 2.3: Structure of FTps<T,N>::multiply

The loop dimensions vary a lot through the whole computation, that means, it can happen that a loop is never executed. In Fig. 2.2 you can see the maximum loop size of each of them. The outer loop is the smallest one and increases linearly according to the order (Listing 2.3). The inner loop has the biggest extent and grows almost exponentially with the order. This behaviour agrees with the observation of Fig. 1.1.



(a) Inner & middle loop. (b) Outer loop.

Figure 2.2: Maximum loop size of FTps<T,N>::multiply for different orders running *sbend*. *Note*: Graphic (a) has a logarithmic y-axis.

The loop construct is numerically optimized such that the innermost loop is only executed if the right hand side of the operation isn't zero. Fig. 2.3 shows the ratio between the number of cycles of the middle loop and the number of calls to the innermost loop. We can see that the higher the order of the system the smaller is the ratio, which means that there are more zeros than non-zeros in the system.

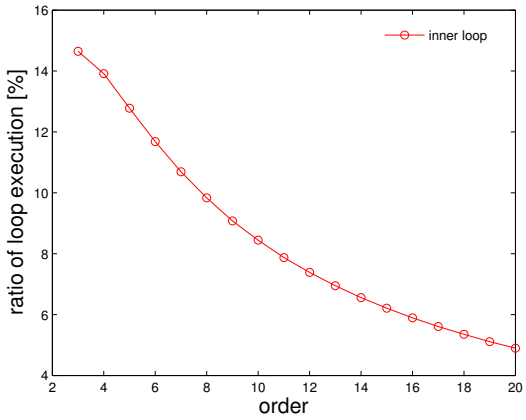


Figure 2.3: Ratio of executions of the inner loop for different orders running *sbend*.

2.1.3 FTps<T,N>::operator+

An advantage of this operator overloading function − which makes the parallelization much easier − are the independent loops. It's used to add two Taylor series componentwise from where we get these stand-alone loops. The function is primary called by the multiply function of FTps, so there's a correlation among them.
The *sbend* program only needs the first five for-loops. The second loop is of special interest because it's never executed in *sbend*, therefore, only four curves are in Fig. 2.4. These grow relatively equal, so the parallelization strategy will be straightforward.

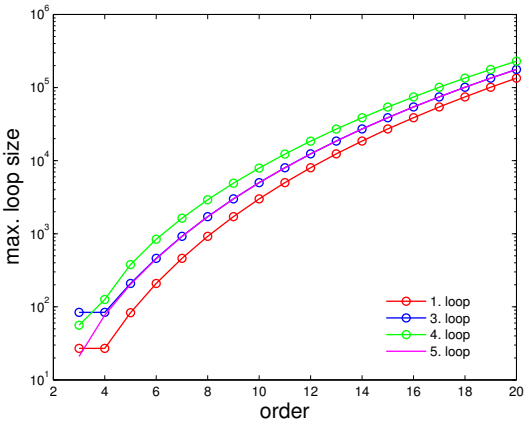


Figure 2.4: Maximum loop size of FTps<T,N>::operator+ for different orders running *sbend*. The curves are again a consequence of the dimensionality of ${}_nD_\nu$ with $\nu = 6$ (section 1.3.3). *Note*: The y-axis is in a logarithmic scale.

2.1.4 FTps<T,N>::derivative

The last function called by ExpMap is FTps<T,N>::derivative. It contains only one while-loop which can easily be transformed to a for-loop (see Listing 2.4 and 2.5). One speedup problem − already in serial execution − is the column-major data access because C++ is developed for row-major order.

```
while(df != dfe) {
  int kp = *product++;
  *df++ = T(FTpsData<N>::getExponents(kp)[var]) * *(f + kp);
}
```

Listing 2.4: While-loop

```
for(T* it = df; it<dfe; ++it) {
  int kp = *product++;
  *it = T(FTpsData<N>::getExponents(kp)[var])* *(f + kp);
}
```

Listing 2.5: For-loop

Another difficulty is the assignment to the variable kp from the array product in the second line of Listing 2.4 and Listing 2.5, because it's needed as index variable in the function call. Therefore, it has to be considered that the loop is kept in the same order otherwise the outcome of the operation is written to the wrong address of the result vector (Fig. 2.5).

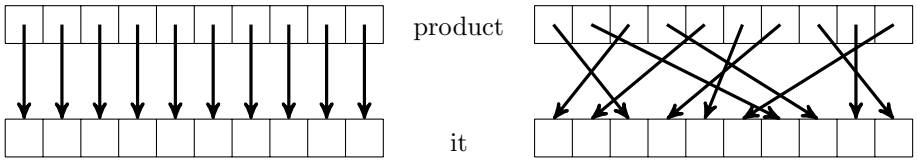


Figure 2.5: Data access in correct order (left) and wrong order (right).

The loop extent increases in an exponential manner (Fig. 2.6). These derivatives are used for Taylor series expansions, so the growth with the order in consideration with formula (1.21) is reproducible.

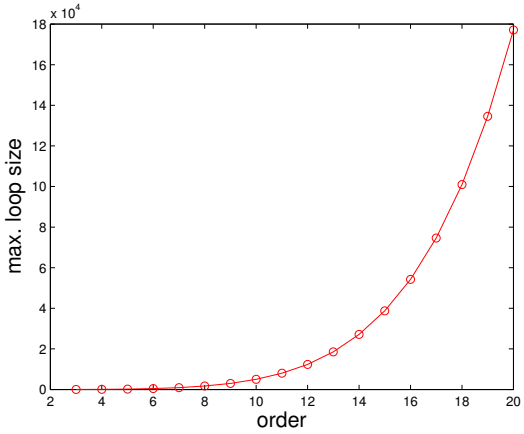


Figure 2.6: Maximum loop size of FTps<T,N>::derivative for different orders running *sbend*.

2.2 Methods

The various parallelizing attempts are treated in this part. It's split into four subsections discussing the already known functions and a further approach that can be applied to all three functions.

2.2.1 FTps<T,N>::multiply

By a reason of efficiency the first attempt was to create all threads outside the nested loop construct. Threads constructed inside a loop are created in every iteration step. This can lead to an overhead bigger than the computation with a negative effect of slowing down.
Unfortunately this approach led a program run crash with a segmentation fault (more precisely: core dumping). The dynamic modification of the increment variable of the middle loop induces the problem of a race condition which couldn't be solved without serializing the code.
The result of Fig. 2.3 and the fact that the innermost loop has the biggest extent (Fig. 2.2) legitimate the strategy to only parallelize the innermost loop, because the overhead won't be the predominant part.
All further particulars of the different inner loop parallelization versions aren't discussed in this section because they only differ in their scheduling strategies and parallelization conditions.

2.2.2 FTps<T,N>::operator+

One idea to parallelize this function is to use the OpenMP section-construct (see Listing 2.6). One disadvantage is the limitation of active threads inside this construction because only one thread is assigned to every section. The others are waiting at an implicit barrier point at the end of the section-construct. The nowait keyword would make the remaining threads go on and do other work but due to possible race conditions this isn't feasible in our case. Another

disadvantage could be the occurrence of load imbalance − meaning that some threads are waiting while others are still working.
A second more flexible construct relating to the number of active threads is the creation of a parallel block whereat each loop has a #pragma omp for directive (see Listing 2.7). This way the loops are finished sequentially but a single loop is executed by multiple threads which is the opposite of the previous concept.
In the third approach each loop is treated independently as one block (see Listing 2.8). One problem of this procedure could be the overhead of thread creation because all threads are killed and created for every loop. One advantage on the other hand is the usage of if-directives only for certain for-loops whereas in the second approach such a directive is just possible for the whole block, since OpenMP doesn't permit if-directives for #pragma omp for. This kind of directive is very useful to enable or, respectively, disable parallelization.

```
#pragma omp parallel sections
{
 #pragma omp section
 {
  for(/*increment*/) {
   //do something
  }
 }
 #pragma omp section
 {
  for(/*increment*/) {
   //do something
  }
 }
}
```

Listing 2.6: Structure with sections (O1).

```
#pragma omp parallel
{
 #pragma omp for
 for(/*increment*/) {
  //do something
 }
 #pragma omp for
 for(/*increment*/) {
  //do something
 }
}
```

Listing 2.7: Structure with one parallel block (O2).

```
#pragma omp parallel for
for(/*increment*/) {
 //do something
}
#pragma omp parallel for
for(/*increment*/) {
 //do something
}
```

Listing 2.8: Structure without parallel block (O3).

2.2.3 FTps<T,N>::derivative

From section 2.1.4 it's known that data access in the wrong order due to multiple threads constitutes a problem for this function. For keeping the loop iterations in the same sequence OpenMP has a keyword called ordered. The disadvantage is the forced serialization.

```
int kp;
#pragma omp parallel for ordered
for(T* it = df; it<dfe; ++it) {
 #pragma omp ordered
 kp = *product++;
 *it = T(FTpsData<N>::getExponents(kp)[var])* *(f + kp);
}
```

Listing 2.9: Strategy for FTps<T,N>::derivative.

The more threads are running the more communication has to be done to keep the order, so the program will slow down while increasing the number of threads.

2.2.4 Non-Uniform Memory Access (NUMA)

In order to guarantee memory locality to each thread, data initialization has to be controlled. After the first touch policy data is stored to the socket where the thread, which touched the data first, belongs to.
In our case data is allocated in FTpsRep<T,N>::FTpsRep(int, int, int). By the use of a #pragma-directive with static scheduling strategy it's attempted to bind data to a certain thread. With the same directive in the functions the data is then split in the same way to guarantee that each thread does calculations with the data stored on its socket.

Chapter 3

Results

3.1 Cluster Architecture

The scaling measurements were performed on the PSI cluster which is called Merlin4. In the following subsection the high-performance cluster is shown in detail.

3.1.1 Merlin4

The PSI cluster consists in total of 62 nodes where three of them are login nodes. Its characteristics are listed in Tab. 3.1. The benchmarks were executed on nodes with Intel Sandy Bridge CPUs.

#nodes	CPU	RAM	#cores
30	2×Intel Xeon X5670 (2.93 GHz)	16×24 GB 16×48 GB	360
2 (login)	2×Intel Xeon X5670 (2.93 GHz)		24
29	2×Intel Sandy Bridge E5-2670 (2.60 GHz)	64 GB	464
1 (login)	2×Intel Sandy Bridge E5-2670 (2.60 GHz)	64 GB	16

Table 3.1: Node characteristics of Merlin4. [11]

3.1.2 Peak Performance

Using the declarations of [8] and [7], an Intel Sandy Bridge E5-2670 CPU has 166.4 GFLOP/s and an Intel Xeon X5670 CPU has 70.392 GFLOP/s. Their maximum memory bandwith is 51.2 GB/s (E5-2670) [9] or, respectively, 32 GB/s (X5670) [10]. Through formula

$$P_{max} = \min(B \cdot O, P) \tag{3.1}$$

with maximum performance P_{max}, maximum memory bandwith B, operational intensity O and peak floating point performance P [16], the roofline model of a node with two E5-2670 (332.8 GFLOP/s) or two X5670 (140.784 GFLOP/s) respectively, looks like in Fig. 3.1.

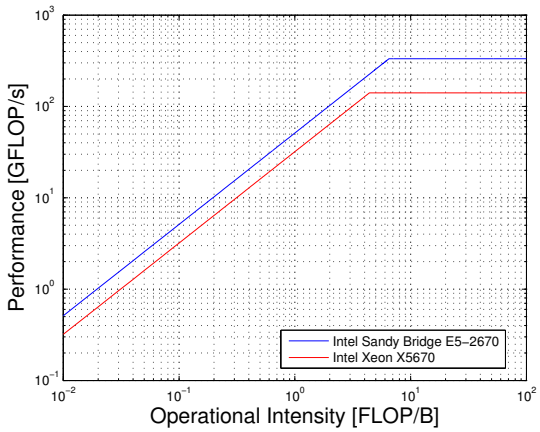


Figure 3.1: Roofline model of Intel Sandy Bridge E5-2670 and Intel Xeon X5670. *Note*: Both axes have a logarithmic scale.

It's easier to reach peak performance on the Intel Xeon X5670 CPU than on the Intel Sandy Bridge E5-2670, since less floating point operations are needed to get into the computationally bound region.

3.2 Measurements

Due to the different number of threads on Merlin4 depending on the type of the CPU, we decided to use nodes with Intel Sandy Bridge CPUs. Benchmarks could be done using up to 16 cores (see Table 3.1) whereat only even numbers of threads were applied. For each measurement a whole node was reserved to avoid disruption by other users. The problem size could be controlled by the tuncation order. The vector space was fixed through *sbend* with $\nu = 6$.
The comparison between different parallelization methods got tested using an order of 20. The choice for this large order depends on the thought to have plausible timings.
After evaluating the best version, the scaling behaviour for smaller orders got explored. For this purpose the following sequence has been chosen: 5, 8, 12, 15, 18 and 20.
Timings were carried out using the OpenMP function `omp_get_wtime()` measuring the wall clock time for a parallelized block. The precision is located around 10^{-9} s according to the return value of `omp_get_wtick()` which is far beyond our demand.
The source code got compiled using the GCC (**G**NU **C**ompiler **C**ollection) compiler g++ 4.7.2 (further details in [6]) without optimization flags − except to `-fopenmp` for the OpenMP library.

3.3 Scalings

The scaling section treats the speedup results of the different versions of each function. For more explanations concerning the implementation of the versions, it's referred to the appendix.

3.3.1 FTps<T,N>::multiply

In the following figures (Fig. 3.3 to 3.5) you can see the speedups of all versions applied to the multiply function. It has to be considered that the y-axis differs among the plots. The versions M10, M12 and M16 in Fig. 3.4 and Fig. 3.5, respectively, indicate the best scaling behaviour. On closer examination it can be recognized that version M10 exhibits the best result.
In an overall observation the speedups stagnate around 8 threads − marked with a red line in the plots.
Versions M7, M13 and M15 are missing in the plots by a reason of reaching the runtime limit. M7 and M13 have runtime and M15 dynamic as scheduling strategy. Using an older compiler version, g++ 4.4.6, these versions indicated no problem (Fig. 3.2).

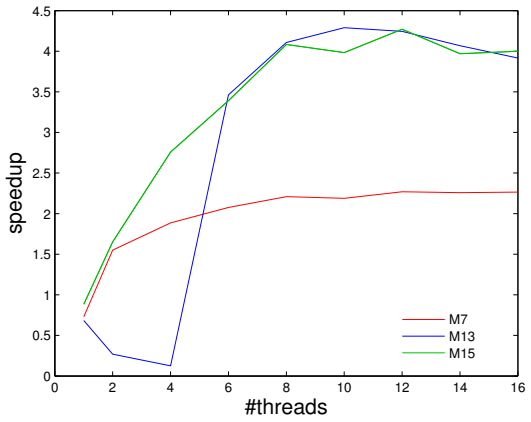


Figure 3.2: Speedup of FTps<T,N>::multiply of versions M7, M13 and M15 using g++ 4.4.6 to compile.

Versions M1 to M7 in Fig. 3.3 and Fig. 3.2, respectively, have the lowest speedup results. Their attribute is an explicit copy constructor of Array1D<T> that is invoked in a parallel region such that the array containing the indices i for the result $\vec{y}_i$ (section 2.1.2) is private to each thread. Version M8 copies this array as well but by the master thread. Through this explicit copying we get these minor speedup results.
Fig. 3.4 shows that by doubling the if-condition for enabling parallelization from M11 to M9, the speedup is halved approximately.
Version M14 has also bad scaling characteristics. The only difference to other versions with much better speedup is the guided scheduling strategy. Thus, this kind of scheduling strategy isn't suitable in that case.

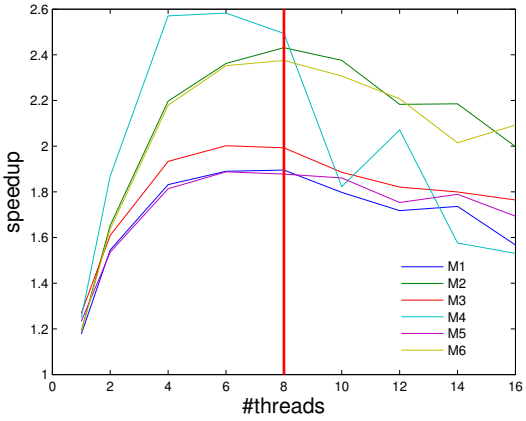


Figure 3.3: Speedup of FTps<T,N>::multiply versions M1 to M6.

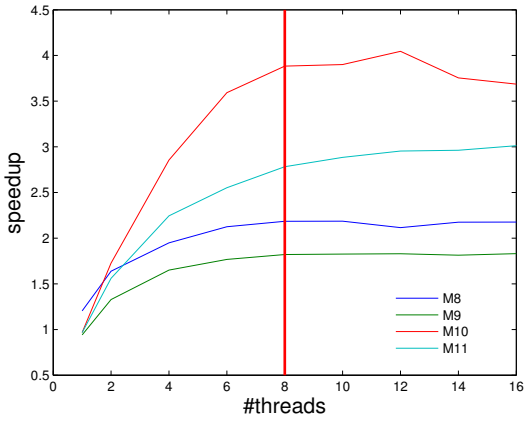


Figure 3.4: Speedup of FTps<T,N>::multiply versions M8 to M11.

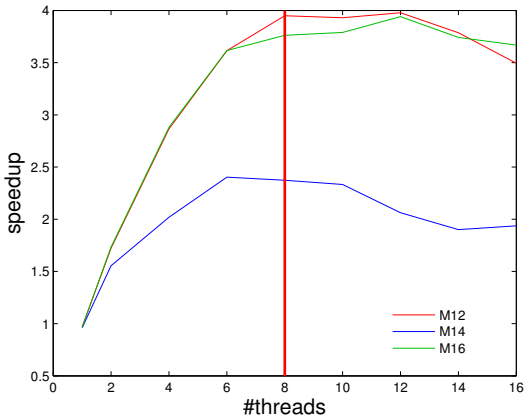


Figure 3.5: Speedup of FTps<T,N>::multiply versions M12, M14 and M16.

Version M10 consists of one #pragma-directive (Listing. 3.1).

```
for(int j = first_g; j < last_g; j++) {
 if(g[j] != T(0)) {
  #pragma omp parallel for
  for(int i = first_f; i < last_f; i++) {
   result[FTpsData<N>::theBook->FTpsData<N>::prod[j][i]] += f[i]*g[j];
  }
 }
}
```

Listing 3.1: Code snippet of version M10.

On a node with Intel Sandy Bridge E5-2670 CPUs there are 2×8 cores. Up to 8 threads only the first CPU is active. For more than 8 threads it's important that the data manipulated by a thread is stored on the same socket. In consideration of this architecture we should gain speedup beyond 8 threads.

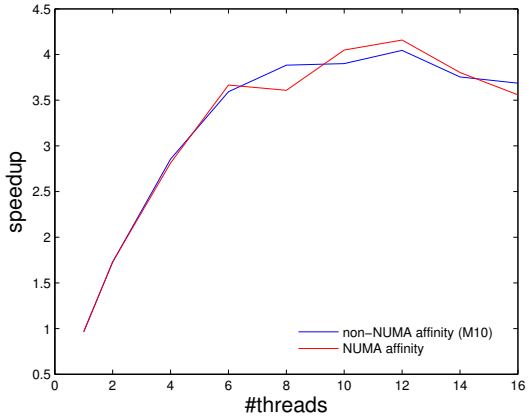


Figure 3.6: Speedup NUMA vs. non-NUMA affinity of FTps<T,N>::multiply.

Despite our attention to memory locality Fig. 3.6 shows no improvement compared to version M10 without NUMA affinity. Another problem that can harm speedup is the memory bandwith limitation.

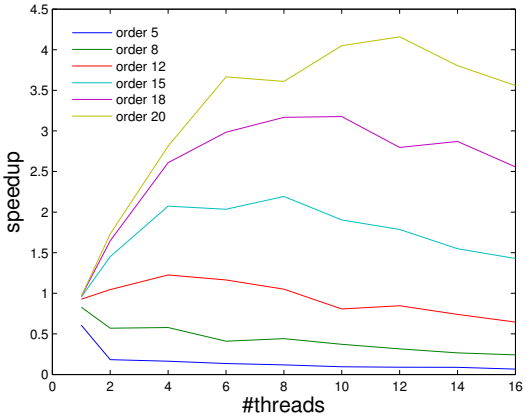


Figure 3.7: Speedup of the NUMA version of FTps<T,N>::multiply with different truncation orders.

After the evaluation of the fastest variant the behaviour for different orders has to be considered. It was decided to take the NUMA variant for this purpose. Fig. 3.7 shows bad scaling characteristics for systems with a truncation order smaller than 13 due to the loop extent. For more and more threads communication predominates execution and the speedup decreases. Through an if-statement in the directive, the parallelization can be disabled for too small problem sizes, thus the effect of decreasing speedup is avoided. It was enabled at a value of 5'000.

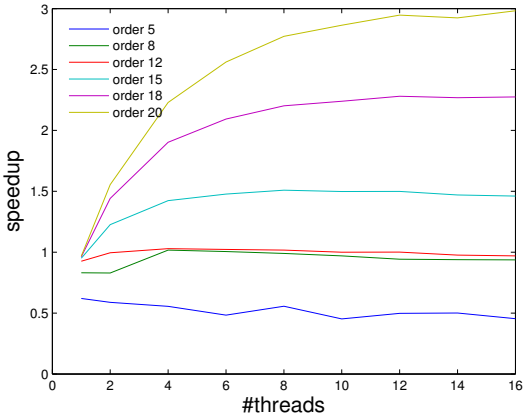
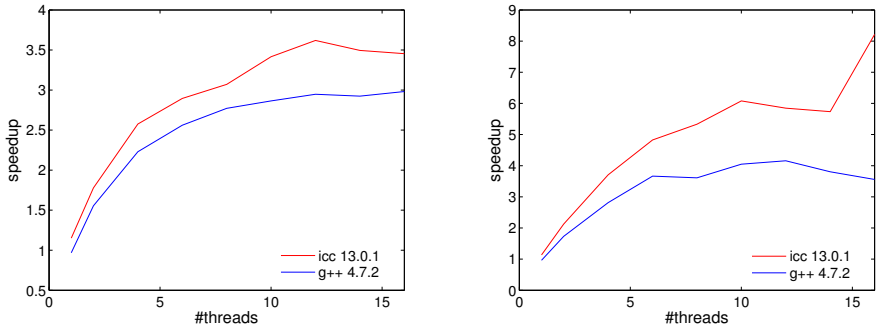


Figure 3.8: Speedup of FTps<T,N>::multiply for different truncation orders with if-directive.

In addition to our speedup problem the directive removed the fluctuations in Fig. 3.7 for higher truncation orders. At second view one recognises an overall loss in speedup. Every time the loop gets executed, the if-directive is checked what diminishes the speedup. For truncation orders of 18 and 20 one loses about

a factor of 1 whereas for an order of 15 the speedup is now 1.5 instead of 2.



(a) NUMA with if-directive. (b) NUMA without if-directive.

Figure 3.9: Comparison between icc 13.0.1 and g++ 4.7.2. Intel Compiler gives better speedup results.

A comparison between the Intel compiler icc 13.0.1 and the GCC compiler g++ 4.7.2 shows that the Intel compiler returns better speedup results. Since the Intel compiler generates executable files using a -O2 optimization level as default, we had to use -O0 -openmp to achieve the same optimization for comparison. The result using icc 13.0.1 with if-directive (Fig. 3.9a) has a similar characteristic compared to g++ 4.7.2. In Fig. 3.9b the speedup with icc 13.0.1 increases until 16 threads whereas the speedup with g++ 4.7.2 doesn't.

Roofline model

In the case of no cache every loop cycle has to read 3 times and write once, according to $\vec{y}_i = \vec{z}_i \cdot \vec{x}_i + \vec{y}_i$ from section 2.1.2. The data type is double, so we have $(3+1) \cdot 8$ byte $= 32$ byte transferred. With an infinite cache size it can be assumed that every value is read only once. Supposed that each value has already been read by other functions, then only 1 write operation is performed and therefore 8 byte are used.
With 3 floating point operations per cycle, we get operational intensities of $\frac{3}{32}$ FLOP/B or $\frac{3}{8}$ FLOP/B, respectively. Using equation (3.1) the algorithm can achieve a maximum performance of 4.8 GFLOP/s with no cache and around 19.2 GFLOP/s with infinite cache. Because these calculations illustrate the extrema, the real performance can only reach a maximum value between 4.8 GFLOP/s and 19.2 GFLOP/s. Therefore, the computation in the innermost loop is definitely memory bound.
With a measurement adding up all loop dimensions of the innermost loop during computation (in total 21'613'432'499) and counting the number of executions (in total 12'651'773), we get an average loop dimension of 1'708. This is the average number of elements which is modified during a loop execution. Through multiplying with the number of floating point operations (here 3) and the number of executions and through dividing the execution time (Tab. 4.3, NUMA version with 12 threads), the performance is around 0.615 GFLOP/s.

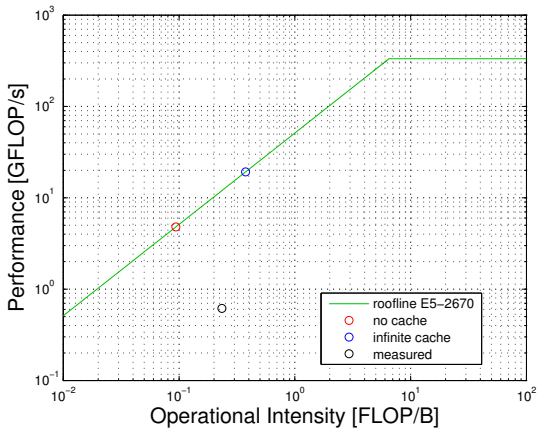


Figure 3.10: Roofline model showing the performance of the innermost loop (measured) using a node with two Intel Sandy Bridge E5-2670 CPUs. The operational intensity is taken to be the mean of both extrema (no cache and infinite cache).

The roofline model in Fig. 3.10 shows that the innermost loop doesn't use the possible performance.

3.3.2 FTps<T,N>::operator+

The speedups of the three approaches, which are already mentioned in section 2.2.2, give the expected behaviour. The section-construct (O1) has almost no speedup at all. Therefore it's not suited for our case. Although the threads in version O3 are instantiated for every loop it exhibits the same scaling characteristics up to 8 threads like version O2.

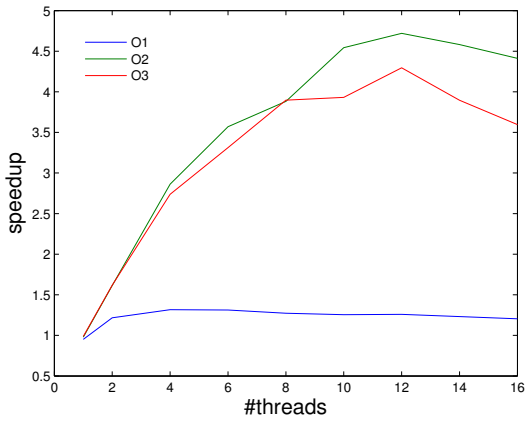


Figure 3.11: Speedup of FTps<T,N>::operator+ for O1, O2 and O3 (see section 2.2.2).

Through different OpenMP scheduling directives it got checked if there's a gain in speedup for the version O2, but Fig. 3.12 indicates no distinctive improvement. The chunk size − needed as second input argument for the scheduling

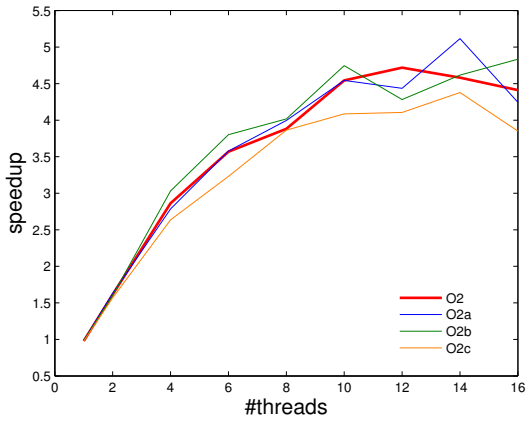


Figure 3.12: Speedup version O2 of FTps<T,N>::operator+ using different scheduling strategies: auto (O2a), static (O2b) and guided (O2c).

algorithms guided and dynamic − was set to the loop extent divided by the number of threads. The red line represents the speedup result of the original version O2 without any specific scheduling. Like in FTps<T,N>::multiply the runs with dynamic and runtime never finished execution. That's why they are missing in Fig. 3.12. After compilation of version O2d with dynamic scheduling using g++ 4.4.6, the execution finished. The result is in Fig. 3.13.

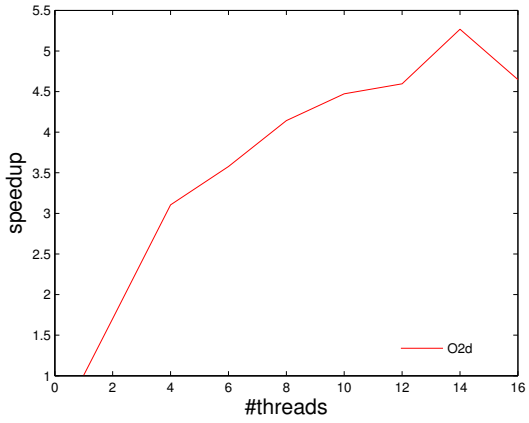


Figure 3.13: Speedup version O2d using g++ 4.4.6 instead of g++ 4.7.2.

A further measurement with attention to the NUMA architecture of the cluster was done whereat it was hoped to get more speedup because of memory locality. Through parallelizing the initialization of the arrays using the static scheduling

algorithm, a chunk, allocated by a specific thread, would reside in memory of the socket where this thread belongs to. This way one can avoid the possible appearance of the more expensive socket overlapping data access. Fig. 3.14 shows the comparison between the measurements with and without NUMA affinity of O2.

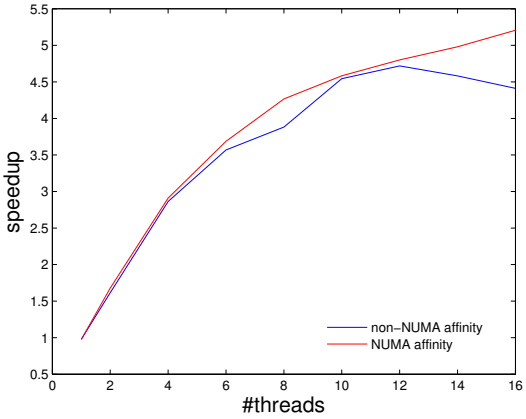


Figure 3.14: Speedup NUMA vs. non-NUMA affinity of O2.

The speedup doesn't increase by introducing the NUMA affinity for less than 12 threads. Nevertheless the code now scales beyond 16 threads.

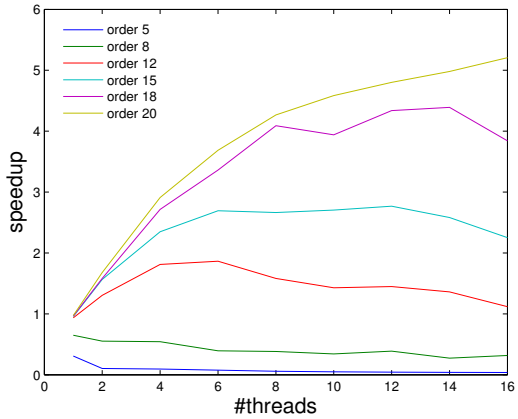


Figure 3.15: Speedup of FTps<T,N>::operator+ for different truncation orders.

For orders smaller than 12 the overhead dominates (see Fig. 3.15), so it's not worth it to parallelize the loops. Therefore, better performance results can be achieved by using an if-directive to enable parallelization at a certain loop extent. Since this directive isn't valid for #pragma omp for, it has to be placed for the whole block. In this case the usage of the loop extent as criteria isn't possible anymore that's why the global truncation order was taken instead. It's called through the function FTps<T,N>::getGlobalTruncOrder().

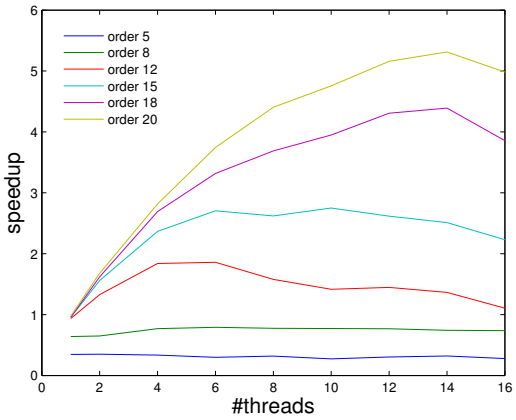
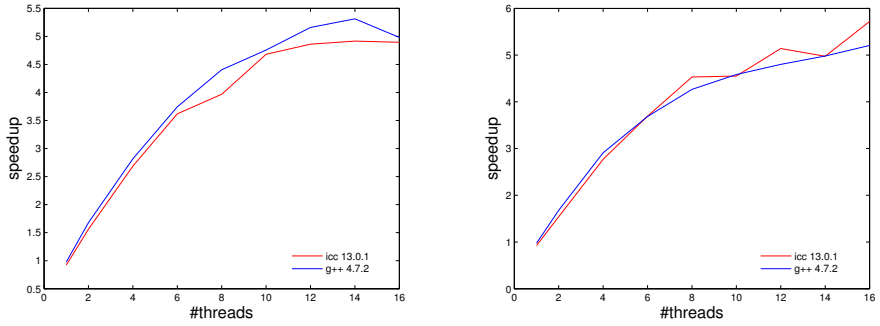


Figure 3.16: Speedup of FTps<T,N>::operator+ with disabling the parallelization for truncation orders smaller than 12.

The speedup doesn't decrease anymore for truncation orders smaller than 12 which can be seen in Fig. 3.16. The blue (truncation order 5) and green (truncation order 8) curves are straight lines. The deviations can be neglected due to measurement inaccuracies.
In contrast to FTps<T,N>::multiply the speedup remains the same after introducing the if-directive.



(a) NUMA with if-directive.

(b) NUMA without if-directive.

Figure 3.17: The measurements with icc 13.0.1 and g++ 4.7.2 have the same speedup characteristic.

In opposite to FTps<T,N>::multiply where we observed better speedup results for the Intel compiler, we have now almost the same speedup results for both. In Fig. 3.17a g++ 4.7.2 is even better than icc 13.0.1. The comparison between Fig. 3.17a and Fig. 3.17b indicates an increase in speedup for icc 13.0.1 whereas g++ 4.7.2 a decrease.

3.3.3 FTps<T,N>::derivative

The timing analysis is done with the various scheduling strategies. For the guided and dynamic strategies the second argument was chosen to be the division of the loop extent by the number of threads.
It was already mentioned in section 2.2.3 that we will not gain much speedup by a reason of the sequential access with ordered. Fig. 3.18 confirms our assumption and even shows a slowing down which is emphasized through the inverse speedup. The more threads are available the more decreases the speedup as a reason of increasing synchronization among the threads.
The scheduling strategy with dynamic leads sometimes to segmentation faults.

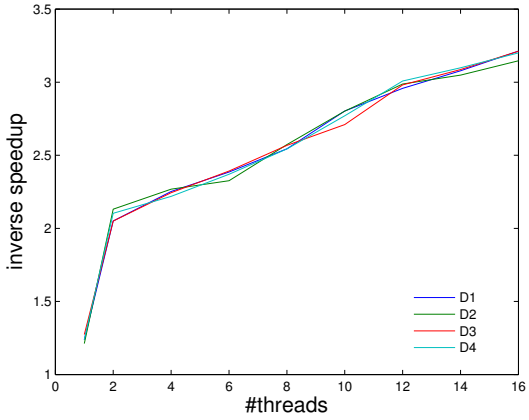


Figure 3.18: Inverse speedup of FTps<T,N>::derivative. The various scheduling strategies: default strategy (D1), auto (D2), static (D3), guided (D4).

Using a NUMA friendly implementation led to the same result (Fig. 3.19). Therefore, it's not advisable to parallelize this function unless the data structures and algorithms are changed.

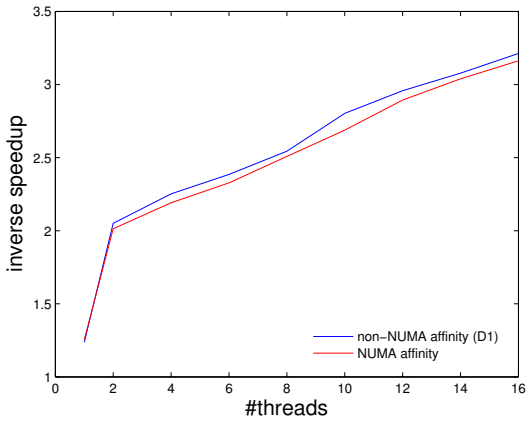


Figure 3.19: Inverse speedup NUMA vs. non-NUMA affinity (D1).

In both figures we see a big decline from one to two threads. Afterwards the gradient stays almost constant.

3.4 Discussion

With profiling the code using GPROF we could evaluate the most time consuming parts and restrict the parallelization to the functions FTps<T,N>::multiply, FTps<T,N>::operator+ and FTps<T,N>::derivative.
The first attempts to parallelize the multiply function resulted in a flattening in speedup initiating from 8 threads. Since an Intel Sandy Bridge E5-2670 CPU has 8 cores we got the suspicion that we have a memory locality problem. Therefore it was tried to ensure that data got allocated on the socket where it would be used. Nevertheless, the speedup was limited around 4 for a truncation order of 20. Comparable measurements using Intel compiler 13.0.1 showed the same effect. Thus a probable hardware limitation can't be excluded. Unfortunately, attempts to measure the memory hit ratio using TAU (**T**uning and **A**nalysis **U**tilities [13]) failed since we couldn't get TAU running. The roofline model shows that this part is memory bound and the maximum possible performance isn't reached.
In FTps<T,N>operator+ we achieved a speedup of about 4.5 with g++ 4.7.2. In opposite to the first function, we have an increasing speedup up to 12 threads. This can be explained by the more simple structure of the operation.
In both cases the truncation order influences the speedup, that is, the smaller the order the less speedup can be achieved. At orders smaller than 12 a parallelization has a negative impact to the execution time. That's a consequence of the shrinking problem size, hence overhead dominates computation.
In the last function we gained no speedup although there's actually enough work to do. It's obvious that the serialization imposed through the keyword ordered prohibits any speedup. The only way of solving that problem is to rewrite the function and changing the underlying data structure such that a serialization can be avoided.
One problem of using OpenMP is the restricted possibility to control data access and storing. Through parallelization of data initialization with OpenMP directives, it can't be guaranteed that each thread finally does computation with its initialized data. Manually threading is perhaps worth it, although it's quite cumbersome and errors are more likely to happen. Alternatively, investigations with compiler flags could probably help optimizing memory locality and probable load imbalances.

3.5 Conclusion

Through profiling and loop analysis, the DA library could partially be parallelized. The scaling results indicate a limitation due to memory locality and cache usage. Further investigations should be done on that.
Depending on the compiler, we reach different maximum speedups ranging from 4 to 6, but the characteristic measured over different numbers of threads remains the same.
Furthermore it has to be paid attention to possible race conditions, although parallelization using OpenMP is quite straightforward.
Continuing works should have a more detailed consideration on the data structure in such a way as to optimize data access for parallelization. In addition the algorithms should be modified − if possible − to get more floating point operations ber byte corresponding to the roofline model result. After these improvements, other parallelization tools should be applied to examine if they yield better speedup results.

3.6 Acknowledgement

Due to the advice of Prof. P. Arbenz who collaborates with Dr. A. Adelmann from PSI, I was drawn attention to this very interesting subject. I have to thank my supervisor Dr. A. Adelmann who helped me during this time. He was my contact person for questions or problems and he made possible to work at PSI which provided me an insight into research. Further I appreciated the support of Dr. T. Kaman and H. C. Stadler Kleeb. Their knowledge concerning OpenMP helped me in improvements and data interpretations.

Chapter 4

Appendix

4.1 FTps<T,N>::multiply

4.1.1 Versions

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   //chunk = 10'000 (M1), 5'000 (M2)
   #pragma omp parallel if(last_f-first_f > chunk)
   {
     Array1D<int> tmp(FTpsData<N>::theBook->FTpsData<N>::prod[j],1);
     #pragma omp for schedule(guided,chunk/omp_get_num_threads())
     for(int i = first_f; i < last_f; i++) {
      result[tmp[i]] += f[i]*g[j];
     }
    }
   }
  }
  //do something
}
```

Listing 4.1: Version M1 and M2.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   #pragma omp parallel if(last_f-first_f > 10000)
   {
    Array1D<int> tmp(FTpsData<N>::theBook->FTpsData<N>::prod[j],1);
    #pragma omp for schedule(auto)
    for(int i = first_f; i < last_f; i++) {
      result[tmp[i]] += f[i]*g[j];
    }
   }
  }
 }
 //do something
}
```

Listing 4.2: Version M3.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   #pragma omp parallel
   {
    Array1D<int> tmp(FTpsData<N>::theBook->FTpsData<N>::prod[j],1);
    #pragma omp for
    for(int i = first_f; i < last_f; i++) {
     result[tmp[i]] += f[i]*g[j];
    }
   }
  }
 }
 //do something
}
```

Listing 4.3: Version M4.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   //chunk = 10'000 (M5), 5'000 (M6)
   #pragma omp parallel if(last_f-first_f > chunk)
   {
    Array1D<int> tmp(FTpsData<N>::theBook->FTpsData<N>::prod[j],1);
    #pragma omp for schedule(dynamic,chunk/omp_get_num_threads())
    for(int i = first_f; i < last_f; i++) {
     result[tmp[i]] += f[i]*g[j];
    }
   }
  }
 }
 //do something
}
```

Listing 4.4: Version M5 and M6.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   #pragma omp parallel if(last_f-first_f > 10000)
   {
    Array1D<int> tmp(FTpsData<N>::theBook->FTpsData<N>::prod[j],1);
    #pragma omp for schedule(runtime)
    for(int i = first_f; i < last_f; i++) {
     result[tmp[i]] += f[i]*g[j];
    }
   }
  }
 }
 //do something
}
```

Listing 4.5: Version M7.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   Array1D<int> tmp(FTpsData<N>::theBook->FTpsData<N>::prod[j],1);
   #pragma omp parallel for if(last_f-first_f > 10000)
   for(int i = first_f; i < last_f; i++) {
    result[tmp[i]] += f[i]*g[j];
   }
  }
 }
 //do something
}
```

Listing 4.6: Version M8.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   //chunk = 10'000 (M9), 5'000 (M11)
   #pragma omp parallel for if(last_f-first_f > chunk)
   for(int i = first_f; i < last_f; i++) {
    result[FTpsData<N>::theBook->FTpsData<N>::prod[j][i]] += f[i]*g[j];
   }
  }
 }
 //do something
}
```

Listing 4.7: Version M9 and M11.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   #pragma omp parallel for
   for(int i = first_f; i < last_f; i++) {
    result[FTpsData<N>::theBook->FTpsData<N>::prod[j][i]] += f[i]*g[j];
   }
  }
 }
 //do something
}
```

Listing 4.8: Version M10.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   // sched_t = auto (M12), runtime (M13), static (M16)
   #pragma omp parallel for schedule(sched_t)
   for(int i = first_f; i < last_f; i++) {
    result[FTpsData<N>::theBook->FTpsData<N>::prod[j][i]] += f[i]*g[j];
   }
  }
 }
 //do something
}
```

Listing 4.9: Version M12, M13 and M16.

```
//do something
for(int gOrd = rhs.itsRep->minOrd; gOrd <= g_max; gOrd++) {
 //do something
 for(int j = first_g; j < last_g; j++) {
  if(g[j] != T(0)) {
   //sched_t = guided (M14), dynamic (M15)
   //nthreads = omp_get_num_threads()
   #pragma omp parallel for schedule(sched_t,(last_f-first_f)/nthreads)
   for(int i = first_f; i < last_f; i++) {
    result[FTpsData<N>::theBook->FTpsData<N>::prod[j][i]] += f[i]*g[j];
   }
  }
 }
 //do something
}
```

Listing 4.10: Version M14 and M15.

4.1.2 Timings

#threads	M1	M2	M3	M4	M5	M6
1	372.672	368.445	345.941	351.164	355.883	369.715
2	283.93	265.41	272.617	234.801	285.668	267.961
4	239.508	199.703	226.926	170.668	241.91	201.328
6	232.152	185.75	219.145	169.871	232.453	186.5
8	231.457	180.465	220.172	175.961	233.637	184.699
10	244.051	184.652	232.656	240.688	235.762	190.16
12	255.422	200.965	240.922	211.82	250.242	198.777
14	252.707	200.746	243.836	278.363	245.184	217.77
16	279.895	219.559	248.684	286.688	259.09	209.656

Table 4.1: Timings (in s) with truncation order 20. Reference time without OpenMP is 438.742 s.

#threads	M7	M8	M9	M10	M11
1	672.566	364.555	466.086	453.098	453.898
2	316.277	267.867	330.27	254.012	281.047
4	260.102	225.078	265.824	153.676	195.438
6	236.277	206.496	248.219	122.121	171.906
8	222.078	200.867	240.922	112.957	157.688
10	224.113	200.75	240.223	112.469	152.125
12	216.145	207.398	239.77	108.445	148.559
14	217.191	201.688	241.875	116.84	148.086
16	216.582	201.609	239.629	119.004	145.668

Table 4.2: Timings (in s) with truncation order 20. Reference time without OpenMP is 438.742 s. Version M7 got compiled using g++ 4.4.6. It's reference time is 490.508 s.

#threads	M12	M13	M14	M15	M16	NUMA
1	452.172	716.668	456.734	555.301	454.891	455.75
2	254.594	1826.5	282.195	297.059	252.98	253.418
4	153.113	3924.7	217.352	177.848	152.328	155.996
6	121.375	141.633	182.566	144.605	121.328	119.676
8	111.086	119.395	184.809	120.141	116.609	121.566
10	111.637	114.332	188.039	123.133	115.785	108.352
12	110.289	115.574	212.723	114.859	111.336	105.516
14	115.859	120.582	230.824	123.555	117.242	115.344
16	125.547	125.234	226.484	122.641	119.594	123.285

Table 4.3: Timings (in s) with truncation order 20. Reference time without OpenMP is 438.742 s. Version M13 and M15 got compiled using g++ 4.4.6. Their reference time is 490.508 s.

#threads	order 5	ord. 8	ord. 12	ord. 15	ord. 18	ord. 20
1	0.0085	0.2115	4.2832	31.2646	168.775	455.75
2	0.0285	0.3071	3.8015	20.5635	98.7129	253.418
4	0.0318	0.3030	3.2427	14.3945	62.3008	155.996
6	0.0384	0.4272	3.4148	14.665	54.4512	119.676
8	0.0438	0.3966	3.7813	13.6162	51.3105	121.566
10	0.0542	0.4710	4.9233	15.6846	51.1562	108.352
12	0.0582	0.5546	4.6943	16.7178	58.125	105.516
14	0.0588	0.6576	5.3674	19.2598	56.627	115.344
16	0.0775	0.7262	6.1577	20.8994	63.5488	123.285

Table 4.4: Timings (in s) with various truncation orders. Reference times without OpenMP in Tab. 4.6.

#threads	order 5	ord. 8	ord. 12	ord. 15	ord. 18	ord. 20
1	0.0084	0.2108	4.2922	31.4082	168.791	453.301
2	0.0088	0.2113	3.9951	24.334	112.75	282.238
4	0.0093	0.1723	3.8618	20.9648	85.4062	196.785
6	0.0107	0.1742	3.8879	20.2061	77.6426	171.297
8	0.0093	0.1770	3.9077	19.7803	73.793	158.27
10	0.0115	0.1807	3.9746	19.9229	72.5664	153.195
12	0.0104	0.1860	3.9724	19.9121	71.25	148.875
14	0.0104	0.1867	4.073	20.3086	71.6367	150.047
16	0.0114	0.1870	4.1001	20.4346	71.4336	147.137

Table 4.5: Timings (in s) with various truncation orders and if-clause. Reference times without OpenMP in Tab. 4.6.

order	time
5	0.0052
8	0.1752
12	3.9749
15	29.8477
18	162.5
20	438.742

Table 4.6: Serial reference timings (in s) with various truncation orders.

	g++ 4.7.2		icc 13.0.1	
#threads	if-clause	no if-clause	if-clause	no if-clause
1	453.301	455.75	528	536
2	282.238	253.418	342	286
4	196.785	155.996	236	164
6	171.297	119.676	210	126
8	158.27	121.566	198	114
10	153.195	108.352	178	100
12	148.875	105.516	168	104
14	150.047	115.344	174	106
16	147.137	123.285	176	74

Table 4.7: Timings (in s) with truncation order 20. Reference for g++ 4.7.2 is 438.742 s and for icc 13.0.1 is 608 s

4.2 FTps<T,N>::operator+

4.2.1 Versions

```
// extents
int for2 = std::max(h_min, g_min);
int for3 = std::max(for2, f_min);
int for4 = std::max(for3, fg_max_min);
int for5 = std::max(for4, f_max);

#pragma omp parallel sections
{
  #pragma omp section
  {
    for(int i=h_min; i < g_min; i++) h[i] = f[i]; //copy below
  }
  #pragma omp section
  {
    for(int i=for2; i < f_min; i++) h[i] = g[i]; //overlap
  }
  #pragma omp section
  {
    for(int i=for3; i < fg_max_min; i++) h[i] = f[i] + g[i]; //do f + g
  }
  #pragma omp section
  {
    for(int i=for4; i < f_max; i++) h[i] = f[i]; //copy above
  }
  #pragma omp section
  {
    for(int i=for5; i < g_max; i++) h[i] = g[i]; //overlap
  }
}
```

Listing 4.11: Version O1.

```
// extents
int for2 = std::max(h_min, g_min);
int for3 = std::max(for2, f_min);
int for4 = std::max(for3, fg_max_min);
int for5 = std::max(for4, f_max);

#pragma omp parallel
{
  #pragma omp for
  for(int i=h_min; i < g_min; i++) h[i] = f[i]; //copy below
  #pragma omp for
  for(int i=for2; i < f_min; i++) h[i] = g[i]; //overlap
  #pragma omp for
  for(int i=for3; i < fg_max_min; i++) h[i] = f[i] + g[i]; //do f + g
  #pragma omp for
  for(int i=for4; i < f_max; i++) h[i] = f[i]; //copy above
  #pragma omp for
  for(int i=for5; i < g_max; i++) h[i] = g[i]; //overlap
}
```

Listing 4.12: Version O2.

```
// extents
int for2 = std::max(h_min, g_min);
int for3 = std::max(for2, f_min);
int for4 = std::max(for3, fg_max_min);
int for5 = std::max(for4, f_max);

#pragma omp parallel for
for(int i=h_min; i < g_min; i++) h[i] = f[i]; //copy below
#pragma omp parallel for
for(int i=for2; i < f_min; i++) h[i] = g[i]; //overlap
#pragma omp parallel for
for(int i=for3; i < fg_max_min; i++) h[i] = f[i] + g[i]; //do f + g
#pragma omp parallel for
for(int i=for4; i < f_max; i++) h[i] = f[i]; //copy above
#pragma omp parallel for
for(int i=for5; i < g_max; i++) h[i] = g[i]; //overlap
```

Listing 4.13: Version O3.

```
// extents
int for2 = std::max(h_min, g_min);
int for3 = std::max(for2, f_min);
int for4 = std::max(for3, fg_max_min);
int for5 = std::max(for4, f_max);

#pragma omp parallel
{
  //sched_t = auto (O2a), static (O2b), runtime (O2e)
  #pragma omp for schedule(sched_t)
  for(int i=h_min; i < g_min; i++) h[i] = f[i]; //copy below
  #pragma omp for schedule(sched_t)
  for(int i=for2; i < f_min; i++) h[i] = g[i]; //overlap
  #pragma omp for schedule(sched_t)
  for(int i=for3; i < fg_max_min; i++) h[i] = f[i] + g[i]; //do f + g
  #pragma omp for schedule(sched_t)
  for(int i=for4; i < f_max; i++) h[i] = f[i]; //copy above
  #pragma omp for schedule(sched_t)
  for(int i=for5; i < g_max; i++) h[i] = g[i]; //overlap
}
```

Listing 4.14: Version O2a, O2b and O2e.

```
// extents
int for2 = std::max(h_min, g_min);
int for3 = std::max(for2, f_min);
int for4 = std::max(for3, fg_max_min);
int for5 = std::max(for4, f_max);

#pragma omp parallel
{
  //sched_t = guided (O2c), dynamic (O2d)
  //nthreads = omp_get_num_threads()
  #pragma omp for schedule(sched_t,(g_min-h_min)/nthreads)
  for(int i=h_min; i < g_min; i++) h[i] = f[i]; //copy below
  #pragma omp for schedule(sched_t,(f_min-for2)/nthreads)
  for(int i=for2; i < f_min; i++) h[i] = g[i]; //overlap
  #pragma omp for schedule(sched_t,(fg_max_min-for3)/nthreads)
  for(int i=for3; i < fg_max_min; i++) h[i] = f[i] + g[i]; //do f + g
  #pragma omp for schedule(sched_t,(f_max-for4)/nthreads)
  for(int i=for4; i < f_max; i++) h[i] = f[i]; //copy above
  #pragma omp for schedule(sched_t,(g_max-for5)/nthreads)
  for(int i=for5; i < g_max; i++) h[i] = g[i]; //overlap
}
```

Listing 4.15: Version O2c and O2d.

4.2.2 Timings

#threads	O1	O2	O3
1	140.98	137.001	135.734
2	110.158	83.0528	82.8232
4	101.763	46.8177	48.9531
6	102.138	37.5486	40.4565
8	105.25	34.528	34.3922
10	106.775	29.4964	34.0914
12	106.448	28.3988	31.199
14	108.838	29.253	34.3996
16	111.285	30.3764	37.2729

Table 4.8: Timings (in s) with truncation order 20. Reference time without OpenMP is 134.025 s.

#threads	O2a	O2b	O2c	O2d	O2e	NUMA
1	135.965	136.081	137.253	137.066	-	137.499
2	82.2662	83.6411	85.4152	81.1345	-	79.8834
4	48.1025	44.2063	50.8398	44.4834	-	46.0583
6	37.4508	35.2569	41.4856	38.615	-	36.3496
8	33.5309	33.3571	34.6798	33.3452	-	31.4079
10	29.4949	28.2424	32.7993	30.8807	-	29.239
12	30.2133	31.3007	32.6367	30.0549	-	27.9129
14	26.1965	29.0323	30.6133	26.2197	-	26.9061
16	31.5786	27.7234	34.8057	29.7174	-	25.7381

Table 4.9: Timings (in s) with truncation order 20. Reference time without OpenMP is 134.025 s. Version O2d got compiled using g++ 4.4.6. It's reference time is 138.134 s.

#threads	order 5	ord. 8	ord. 12	ord. 15	ord. 18	ord. 20
1	0.0046	0.0498	1.0829	8.4835	48.9517	137.499
2	0.0137	0.0584	0.7758	5.1906	29.8286	79.8834
4	0.0150	0.0594	0.5590	3.4589	17.3828	46.0583
6	0.0188	0.0821	0.5430	3.0164	14.0366	36.3496
8	0.0245	0.0843	0.6403	3.0482	11.5393	31.4079
10	0.0291	0.0943	0.7086	3.0025	11.978	29.239
12	0.0329	0.0830	0.6992	2.9344	10.8796	27.9129
14	0.0354	0.1185	0.7441	3.1453	10.7463	26.9061
16	0.0374	0.1019	0.9062	3.6063	12.2874	25.7381

Table 4.10: Timings (in s) with various truncation orders. Reference times without OpenMP in Tab. 4.12.

#threads	order 5	ord. 8	ord. 12	ord. 15	ord. 18	ord. 20
1	0.0041	0.0504	1.0824	8.4932	49.264	137.474
2	0.0041	0.0497	0.7627	5.2067	29.1334	79.8846
4	0.0042	0.0419	0.5504	3.4316	17.5282	47.5788
6	0.0047	0.0408	0.5443	3.0044	14.225	35.7983
8	0.0044	0.0417	0.6415	3.1007	12.7943	30.4241
10	0.0052	0.0418	0.7150	2.9544	11.9522	28.1774
12	0.0047	0.0420	0.6998	3.1037	10.9571	25.9766
14	0.0044	0.0434	0.7420	3.2348	10.7467	25.2261
16	0.0051	0.0438	0.9163	3.6425	12.2395	26.9019

Table 4.11: Timings (in s) with various truncation orders and if-clause. Reference times without OpenMP in Tab. 4.12.

order	time
5	0.0014
8	0.0323
12	1.0124
15	8.1221
18	47.191
20	134.025

Table 4.12: Serial reference timings (in s) with various truncation orders.

#threads	g++ 4.7.2 if-clause	g++ 4.7.2 no if-clause	icc 13.0.1 if-clause	icc 13.0.1 no if-clause
1	137.474	137.499	145.188	144.688
2	79.8846	79.8834	85.625	86.6875
4	47.5788	46.0583	49.75	48.25
6	35.7983	36.3496	36.9375	36.1875
8	30.4241	31.4079	33.6875	29.5
10	28.1774	29.239	28.5625	29.375
12	25.9766	27.9129	27.5	26
14	25.2261	26.9061	27.1875	26.875
16	26.9019	25.7381	27.3125	23.375

Table 4.13: Timings (in s) with truncation order 20. Reference for g++ 4.7.2 is 134.025 s and for icc 13.0.1 is 133.688 s

4.3 FTps<T,N>::derivative

4.3.1 Versions

```
int kp;
#pragma omp parallel for ordered
for(T* it=df; it < dfe; ++it) {
  #pragma omp ordered
  kp = *product++;
  *it = T(FTpsData<N>::getExponents(kp)[var]) * *(f + kp);
}
```

Listing 4.16: Version D1.

```
int kp;
//sched_t = auto (D2), static (D3)
#pragma omp parallel for ordered schedule(sched_t)
for(T* it=df; it < dfe; ++it) {
  #pragma omp ordered
  kp = *product++;
  *it = T(FTpsData<N>::getExponents(kp)[var]) * *(f + kp);
}
```

Listing 4.17: Version D2 and D3.

```
int kp;
//sched_t = guided (D4), dynamic (D5)
//nthreads = omp_get_num_threads()
#pragma omp parallel for ordered schedule(sched_t,(dfe-df)/nthreads)
for(T* it=df; it < dfe; ++it) {
  #pragma omp ordered
  kp = *product++;
  *it = T(FTpsData<N>::getExponents(kp)[var]) * *(f + kp);
}
```

Listing 4.18: Version D4 and D5.

4.3.2 Timings

#threads	D1	D2	D3	D4	D5	NUMA
1	5.1627	5.0535	5.3116	5.1651	-	5.2306
2	8.5520	8.8872	8.5479	8.7699	-	8.3951
4	9.3922	9.4592	9.3569	9.2533	-	9.1400
6	9.9459	9.7007	9.9764	9.8843	-	9.7051
8	10.6092	10.7304	10.7094	10.6075	-	10.4606
10	11.6894	11.685	11.3014	11.5524	-	11.2111
12	12.3341	12.4584	12.4333	12.545	-	12.0697
14	12.8366	12.7123	12.8729	12.9237	-	12.6751
16	13.3966	13.1214	13.3954	13.3474	-	13.1899

Table 4.14: Timings (in s) with truncation order 20. Reference time without OpenMP is 4.1701 s.

Bibliography

[1] A. Adelmann et al. "The OPAL Framework User's Reference Manual". In: Version 1.2.0 (2013). PSI. URL: http://amas.web.psi.ch/docs/opal/opal_user_guide.pdf.

[2] Andreas Adelmann. *Particle Accelerator Physics & Modeling I - Lecture 1*. ETH Zurich. 2013.

[3] Andreas Adelmann. *Particle Accelerator Physics & Modeling II - Lecture 2*. ETH Zurich. 2013.

[4] Andreas Adelmann. *Physical Methods used in OPAL*. PSI. 2013.

[5] Martin Berz. *Modern Map Methods in Particle Beam Physics*. Academic Press, 1999. ISBN: 0-12-014750-5.

[6] GCC. *GCC, the GNU Compiler Collection*. 2013. URL: http://gcc.gnu.org/.

[7] Intel. *Intel Xeon Processor 5600 Series*. 2011. URL: http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf.

[8] Intel. *Intel Xeon Processor E5-2600 Series*. 2012. URL: http://download.intel.com/support/processors/xeon/sb/xeon_E5-2600.pdf.

[9] Intel. *Intel Xeon Processor E5-2670*. 2013. URL: http://ark.intel.com/products/64595/Intel-Xeon-Processor-E5-2670-20M-Cache-2_60-GHz-8_00-GTs-Intel-QPI.

[10] Intel. *Intel Xeon Processor X5670*. 2013. URL: http://ark.intel.com/products/47920/.

[11] Valeri Markushin. *Local HPC User Guide for Merlinl4 Cluster*. 2013. URL: https://intranet.psi.ch/PSI_HPC/LocalHpcUserGuide#What_is_Merlin4_63.

[12] OpenMP and R. Friedman. *OpenMP*. 2013. URL: http://openmp.org/wp/.

[13] University of Oregon, Los Alamos National Laboratory, and Research Centre Julich ZAM Germany. *Tuning and Analysis Utilities*. 2013. URL: http://www.cs.uoregon.edu/research/tau/home.php.

[14] J. Osier et al. *GNU gprof - The GNU profiler*. 2013. URL: http://ftp.gnu.org/pub/old-gnu/Manuals/gprof-2.9.1/html_node/gprof_toc.html.

[15] AMAS PSI. *Accelerator Modelling and Advanced Simulations (AMAS)*. 2013. URL: http://amas.web.psi.ch/.

[16] Samuel Williams, Andrew Waterman, and David Patterson. “Roofline: An Insightful Visual Performance Model for Multicore Architectures”. In: *Communications of the ACM* 52.4 (2009), pp. 65–76.