11th World Congress on Computational Mechanics (WCCM XI)
5th European Conference on Computational Mechanics (ECCM V)
6th European Conference on Computational Fluid Dynamics (ECFD VI)
E. Oñate, J. Oliver and A. Huerta (Eds)

OPTIMIZING THE MEMORY ACCESS PERFORMANCE OF FASTEST'S SIPSOL ROUTINE[1]

Michael Burger* **and Christian Bischof***

* TU Darmstadt, Mornewegstrasse 30, 64293 Darmstadt, Germany
e-mail: {michael.burger, christian.bischof}@sc.tu-darmstadt.de, web page:
www.sc.tu-darmstadt.de

Key words: Data Structures, Performance Optimization, Parallelization, Cache Optimization

Abstract. In this article the runtime behavior of the simulation software Fastest is investigated and its performance is optimized. The main performance bottleneck of Fastest is the sipsol subroutine. An analysis shows that the memory accessing behavior is the main reason for the high runtime. As a consequence, a new data structure is developed which on the one hand increases the speed but on the other hand preserves the ability to parallelize the calculations on the data grid executed within the sipsol subroutine. The ratio between calculation and memory reads/writes is improved significantly, and the performance in certain cases doubled. Finally, approaches to efficiently implement a parallelization within the new structure are proposed.

1 INTRODUCTION

1.1 Fluid simulation Software Fastest

Fastest is a program for the calculation of flows within 3D environments. It was initially developed at the University of Erlangen-Nuernberg [1]. At the moment, there exist different versions of the code at different institutions like at the Friedrich-Alexander-University Erlangen-Nuernberg and the TU Darmstadt. Fastest offers the user the possibility to simulate free topologies within the used multigrid approach. Because of its high flexibility Fastest is in widespread use [1]. For scientific use the Fastest code can be obtained for free. For this paper we employ the version and test case in the main development branch of the Institute of Numerical Methods in Mechanical Engineering (FNB) at TU Darmstadt. It uses a four grid level approach, where level 1 represents the coarsest and level 4 represents the finest one. The solution for the flow equations is based on the iterative

[1] Parts of this work were funded by the German Research Foundation (DFG) under Grant No. GZ. BI714/5-1 (SPPEXA/CATWALK) and the Hessen State Ministry of Higher Education, Research and the Arts under "Foerdungsbuchungskreis 2995"

SIMPLE (Semi-Implicit Method for Pressure Linked Equations) algorithm developed by Spalding and Patankar [2].

The investigated version uses an implicit time stepping method. The implemented algorithms are block-based and parallelized with the Message Passing Interface (MPI).

1.2 Related Work

The Strongly Implicit Solver (sipsol) of Fastest was also investigated and optimized by Scheit et al [3]. They reduced the workload by removing unnecessary calculations and taking advantage of the symmetry of the stiffness matrix. They additionally reduced the resolution of the used data types within the solver from double to single precision. This exchange is possible as this precision loss does not influence the convergence behavior and the final result. Scheit et al also improved the MPI communication by changing from synchronous to asynchronous MPI library functions. In the original version, lots of time was spent on waiting for other processes, which were still in calculation. This led to a quasi-serialization of this program part and resulted in the fact that more time was spent for communication than for computation within the solver. For details see [3] section IIIB.

Investigation of the impact of different data structures on the performance of algorithms was done by Kreutzer et al [4]. The authors compared the performance of different data structures for the problem of sparse matrix vector multiplication. With Intel Sandy Bridge and Xeon Phi as well as NVidias Kepler K20 they took three modern architectures into account. They show that not every data structure is suitable for all architectures and that the choice of the format massively influences the runtime. They also suggest a data structure called SELL-C-σ which, when suitably parametrized, performance well on all these platforms. Leung and Zahorjan [5] dealt with the question how arrays can be restructured in the case of a given access pattern with the aim to optimize spatial locality.

2 METHODOLOGY

2.1 Test Environment

The runtime behavior of the code was mainly tested on the Interimscluster of TU Darmstadt. The used nodes are equipped with four AMD Opteron 6238 (Barcelona) processors with 12 cores each. They possess a three level cache hierarchy with a 16 MiB level 3 cache. These cores have access to 64 GB of main memory.

The results have been verified on a two socket system with Intel E5-2670 16 core processors. They also employ a three leveled cache hierarchy with 20 MB at the lowest level.

All measurements have been done on unused nodes with a fixed binding of threads to one processor core. The Intel Cluster Studio in version 13.1 was used. The optimization level was set to full optimization (Ox).

For the investigation of the runtime behavior of Fastest, several software tools were

used, namely gprof , the valgrind tool suite with its tools callgrind and cachegrind, Vampirtrace and the Intel VTune amplifier. With their help it can be shown that the subroutine sipsol executes a high percentage of the overall total instructions during a program run. For the configuration which takes the grid level 4 into account, over 50 % of all instructions are invoked within sipsol. Table 1 summarizes the values which were gathered from callgrind.

2.2 Performance Analysis of the Code

Grid	Total	Sipsol	Percentage Sipsol
Level 1	$7.3 * 10^9$	$2,3 * 10^9$	$31,5\%$
Level 2	$1,1 * 10^{11}$	$0,48 * 10^{11}$	$43,6\%$
Level 3	$6,7 * 10^{11}$	$3,1 * 10^{11}$	$46,2\%$
Level 4	$4,3 * 10^{12}$	$2,3 * 10^{12}$	$53,5\%$

Table 1: Distribution of executed instructions during a run of Fastest on the actual grid level only. Values are the output of the tool callgrind from valgrind suite.

sipsol is an implementation of a "'Strongly Implicit Solver"' and is based on the work of Stone [6]. It is contained in two different versions within the corresponding code file. The first one, in the following called ijk-version, proceeds the data elements along the three coordinate axes via a nested loop construct. The elements are saved in a linearized 1D array.

The second version, in the following called vector-version, has another data processing scheme based on 3D diagonals which is explained in detail in section 2.3. It is also stored in a one dimensional array. Since the vector-version allows to parallelize the run through the grid, it is the one which was optimized for this paper.

The central position of sipsol becomes clear when one looks at the execution flow of Fastest. Because of the usage of the already mentioned SIMPLE algorithm, the system consists of an outer and an inner iteration, e.g. an outer and an inner solver. So after the initialization phase the subroutine solvense for solving the Navier-Stokes equations is invoked. Within this subroutine there are always alternated calls of the procedures calcuvw and calcp, which determine velocities and the pressure in the actual step respectively. calcuvw calls sipsol separately for all three spatial directions u, v and w. On the other side calcp invokes sipsol only once, but with a higher number of maximum iterations. This procedure is shown in figure 1.

The structure of the sipsol routine itself is illustrated in Figure 2. The initialization (*01* in Figure 2) is followed by the LU decomposition (*02*) of the input matrix. Then the code enters the so called inner loop (*03*), where the actual computation takes place. This loop is repeated a fixed number of times (steps), which is set via a parameter of the sipsol call. The inner loop is divided in two main parts. First is the computational part. It consists of a forward substitution step (*03a*), the calculation of the residual (*03b*) and

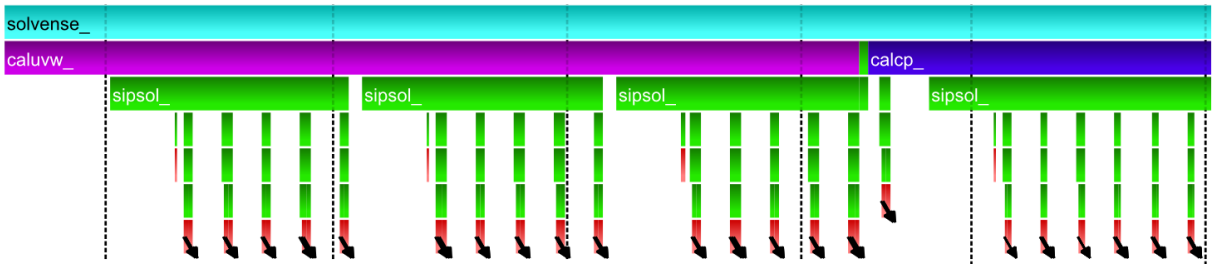

Figure 1: **Snapshot of the Execution Flow of Fastest**: The routines calcuvw and calcp are called in an alternating fashion until the simulation ends. These two routines are the callers of the strongly implicit solver. The figure is a modified screenshot from Vampirtrace. It does not show the fact, that calcp invokes sipsol with a maximal iteration count of 25, while every call from caluvw uses only a maximum of five.

the backward substitution (*03c*). During these steps a lot of floating point operations take place. These operations are executed within two loops over all elements which are part of the considered block. In the second part of the inner loop the borders of the grid are exchanged with its neighbors (*03d*) and the next iteration is started. After the last iteration sipsol returns the results to its caller (*04*).

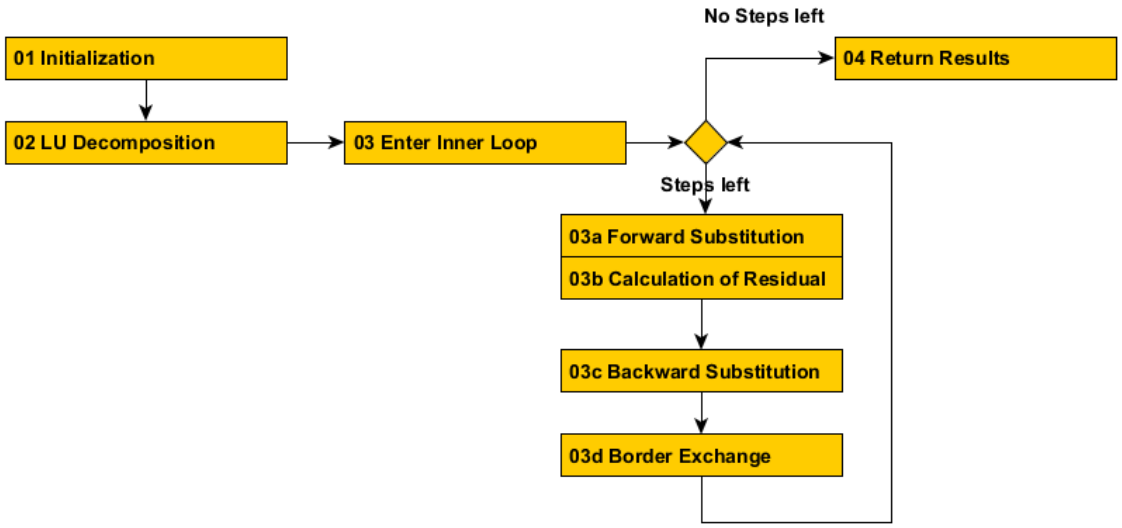

Figure 2: **Flow Diagram of Sipsol Routine:** Details are given in the text.

2.3 Memory Access Pattern Issues

The main problem during the execution of sipsol is its memory access pattern within the computational loops. The elements of the grid are ordered in RAM by their 3D positions running through the grid along the three coordinate axes like it was considered in Listing **??**. So the memory reads during this processing scheme are all consecutive.

But because of the five-point-stencil method in the three dimensional space, the following element in this permutation is dependent on the result of the evaluation of its predecessor.

In order to parallelize such a pass through the grid with threading libraries like OpenMP or Intel Cilk Plus, code changes would be needed. The first possibility is to use a red-black ordering scheme for the elements in the 3D grid. For a large amount of elements there are also problems with memory accesses. If the grid does not fit into cache, every element has to be read several times. To overcome this, blocking can be used to increase cache reuse. But this introduces more communication for border exchanges between the blocks or changes the numerical behavior of the calculations.

The way to easily and directly parallelize the sipsol code with the underlying memory layout and loop structure would be to duplicate the data arrays. One group of the data arrays is used to collect the result values while the other one is used as input. Then for every iteration the roles for both arrays are exchanged. But because of its high memory demand this approach is not suitable for Fastest.

Instead the 3D diagonal scheme of the vector-version is the base for our work. In this scheme the 3D grid is partitioned into subsets, which correspond to diagonals in 2D. They can be thought of as skew cutting planes through the grid elements which are ordered in a cuboid. Figure 3 shows how these subsets look like. The first 3D diagonal is a single element at the corner of the cuboid shaped space. The second one, shown in brown, consists of three elements which are the direct neighbors of the single element from the first diagonal. The lasts one which can be seen in figure 3 is the third diagonal which is formed by six elements and visualized in green. Around the actual grid there lies a border of ghost cells with thickness 1. These ghost cells are exchanged between the different processes after each iteration of the inner loop. So the first blue element in figure 3 is only dependent on three ghost cells.

To identify the diagonals, the numbering scheme for the first three diagonals is continued for all following ones, and so each diagonal that covers its predecessor gets an index incremented by one.

With increasing number of the diagonals the number of elements within a diagonal increases. The dependence of the diagonal's number and its elements can be described by the formula

$$n_{elements} = elements_{lastdiagonal} + i_{diagonal}$$

where $n_{elements}$ is the number of elements in the considered diagonal, $elements_{lastdiagonal}$ is the amount of elements of the predecessor diagonal and $i_{diagonal}$ is the index of the considered diagonal. After some growth the size of the diagonals reaches a maximum and then decreases in the same manner as it has increased before. The length of the maximum diagonal is dependent on the shape of the grid. For a cube there is only one longest diagonal, but for cuboids this value varies. The longer the cuboid, the more diagonals with maximum length are contained in it.

These geometric entities have the property that all contained elements can be calculated independently of the others. So is possible to parallelize the calculations within

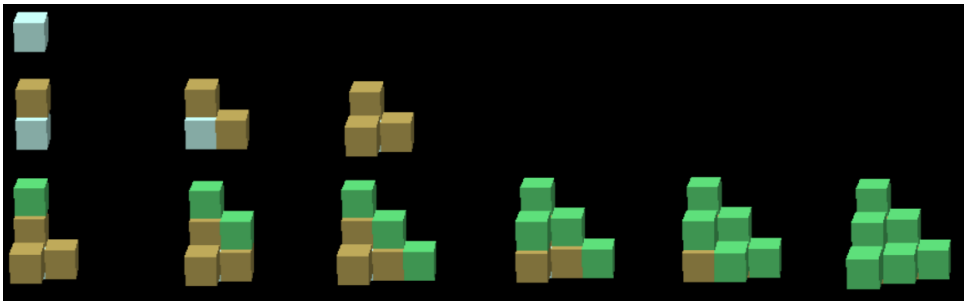

Figure 3: **3D Diagonals:** The first three 3D diagonals when starting the processing from the element at the corner of the grid.

Table 2: A Simple Example Table

npoi:	**ijk:**	**npoi:**	**ijk:**
202610	158879	202615	160244
202611	158944	202616	160309
202612	160049	202617	160374
202613	160114	202618	160439

the diagonals. Listing 1 shows the code which processes the elements in the described order. The index *ndia* runs over the individual diagonals of the grid, while the index *npoi* identifies a single element within a diagonal. The npoi index is unique for the grid, so that an item is unambiguously identifiable by its *npoi* index.

```
1  do ndia=1,numdia(ngr,m)
2    do npoi=npsta(ndia,ngr,m)+1,npsta(ndia+1,ngr,m)
3      ijk=ijkdia(npoi) ! transform index to 3D position
4      ! do calculation on element with index ijk
5    enddo
6  enddo
```

Listing 1: Pseudocode for data processing in vector-version sample

But this method has a major drawback. While the memory reads in the 3D grid processing scheme are consecutive, the 3D diagonal scheme leads in nearly all cases to jumps in memory when the next element is processed. The reason can be seen in line 3 of listing 1. The loop index *npoi*, which is consecutive, is transformed to the 3D position of the associated grid element. Table 2 lists the corresponding ijk-indices for four succeeding loop iterations over *npoi*.

Large is the jump over more than 1000 elements between $npoi = 202611$ and $npoi = 202612$. Those jumps always occur when a successive element lies in another row than its

predecessor. Even larger jumps are observed when the actual horizontal plane switches between to elements switches. That means they lie at a different height within the cuboid. But still the average jump between two elements exceeds 50 items and lies beyond the size of a cache line. So these data accesses which occur during the inner loop lead to a lot of cache misses. Additionally neither the compiler nor the hardware is able to use prefetching. This is due the fact, that no regular memory access pattern exists.

Without prefetching and exploitation of caches, the routine often has to wait until requested data from memory has been transferred and calculation cannot proceed. For the same reason, the vectorization units of modern hardware (AVX in Intel Xeon processors or the 512bit SIMD unit of Intel Xeon Phi coprocessor cards) can not work efficiently, because they also have to wait for the data. Altogether this leads to a slow single core execution time, that is addressed by our optimization.

3 OPTIMIZATION STEPS

3.1 A new Layout for the Data in Memory

To overcome the memory accessing issue described in section 2.3, an appropriate layout for the data must be developed. The approach works as follows: Each 3D diagonal is split into lines along one of the coordinate axis. Each line is then projected into a quadratic, fixed sized 2D grid in memory. So for each diagonal there exists a 2D quadratic grid in memory. Figure 4 visualizes this process.

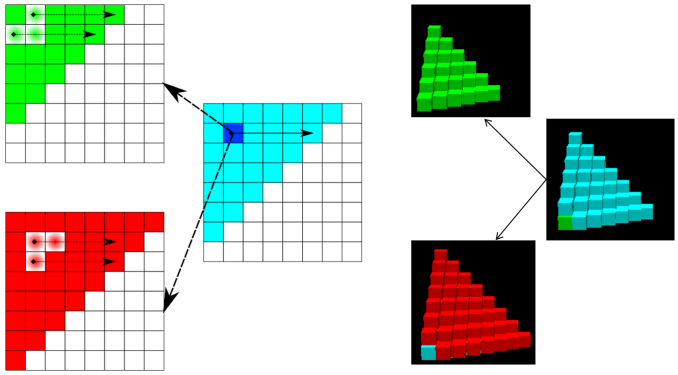

Figure 4: **New Memory Layout:** 3D diagonals are projected on 2D grids. Details are given in the text.

On the right side three adjacent diagonals are show from top to bottom. To illustrate that they cover their predecessor, for the blue and the red one, the last element was omitted to indicate the underlying green and blue diagonals respectively. Now assume that the coordinate system is oriented as indicated in the picture. The y-axis is the one that is pointing out of the image plane. One can now count the lines built by the elements from left to right. The left most line consists of six elements and is projected as the first row of the 2D grid on the left upper part in the figure. The next one has five elements,

and becomes the second row in 2D. This process is repeated for every line in the green diagonal and for every diagonal in the grid.

Figure 4 also indicates the advantage of this new scheme. For example to calculate the dark blue item in the middle of figure 4, the program has to access its two bordering diagonals. In the case of Figure 4 these are the green and the red ones. The elements which need to be read are marked by the pale colors. In the next step the item right to the dark blue one must be processed. To calculate it, the same lines of the green and red diagonal must be accessed and the successional elements of the previous steps are needed. These elements reside with high probability already in cache and can be used immediately without memory transfers.

In addition, the index variable for all involved fields only consists of the loop variable and a constant offset. This allows the compiler to use auto-vectorization, which was tested with Intel Fortran Composer 13.1 and gcc 1.6. Both compilers recognize that there are no dependencies between loop iterations and vectorize the loop body.

Another way to improve the efficiency of vectorization is the extraction of calculation loops into separate subroutines. Their only parameters are a pointer to the beginning of the data vector and the length of the data portion of interest. For this type of functions the compiler can easier recognize the independence of the single iterations during the calculation loop and generate auto-vectorized code.

But there is a drawback to the modified code: The new layout has to be instantiated with every call of sipsol and the input data has to be moved from the old memory position to the new one. This step has to be reversed at the end of the routine, when the data is moved back from the new structures to the returned result vectors which must be structured in the old way. Additionally there is an overhead because Fastest partitions the grid into blocks and calculates these blocks separately. This requires border exchanges between the seperate blocks which have to occur in the old structure. So at the end of each iteration the borders must be copied to the old layout, are exchanged between the blocks and the updated borders are integrated in the new scheme before the next iteration starts. The amout of overhead is analysed in the following chapter 4.

4 RESULTS

As described in section 3, the new data layout was designed to optimize the memory access behavior of the application. Memory reads are now consecutive within the diagonals and this results in efficient usage of the cache hierarchy and the vector units of modern hardware architectures. Figure 5 visualizes the performance difference of the original version and the optimized code for a different number of iterations of the inner loop when using the finest grid level 4.

It can be seen that with the new version it is possible to nearly double the performance e.g. to half the runtime for grid level 4 calculations. Even for the smallest tested iteration count of 25 there is a performance improvement of nearly 70 %. For a higher number of iterations the performance increase is nearly 90 %. The lower increase in the case

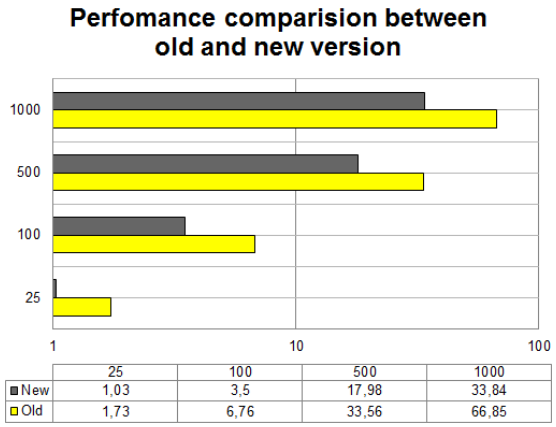

Figure 5: **Performance Results with the new Layout**: The y-axis shows the amount of inner loop iterations on grid 4. The x-axis shows the runtime in seconds and is scaled logarithmically. The new layout shows a significant performance gain when compared to the initial vector-version.

	500 iterations	25 iterations
Time for datastructure	0,8 s (0,12 + 0,68)	0,38 s (0,11 + 0,27)
Time for calculation	35,0 s	0,55 s
Ratio data. / compu.	0,023	0,69

Table 3: **The Relation of Computation and Management**: The first row shows the overall time for creating and maintaining the new data structure. The two values in brackets correspond to the initial creation and the updating respectively. Row two summarizes the time which was spent in the three computation loops, while the last row gives the ratio between time for computation and for data structure management.

of 25 iterations results from the overhead of instantiating the new data layout and the rearrangement of data before entering the inner loop. For smaller iteration counts the benefit of the new scheme will even be less. But for higher iteration counts this overhead is negligible as can be shown be investigating the sipsol routine with Intel VTune Amplifier on two test cases. In the first one Fastest enters sipsol, executes the inner loop with 500 iterations and then terminates programm execution. The second case does the same with a maximum iteration count of 25. Then several Advanced Hotstops Analyses have been carried out and their results have been averaged. Table 3 shows the results.

It is apparent that the higher iteration count leads to a much better computation to data structure management ratio. Both ratios differ by a factor of 30. But important to notice that the second value within the bracket is less during the 50 iterations. This is because of the border exchange at the end of each loop iteration where the border values have to be copied from the new structure to the new one and reverse. If the iteration count is lower, the accumulated time for border exchange preparations will also be lower.

As an additional method to investigate the performance of the layout, hardware coun-

ters have been used to investigate the cache behavior. To this end, the old vector-version of sipsol was sampled against the new one with the VTune Amplifier. Both executed 500 iterations of the inner loop before the program was terminated. *Memory Access* for Sandy and Ivy Bridge systems was chosen as analysis type. The results confirm the theory behind the new data scheme. The amount of Last Level Cache Misses (in this case Level 3), drops from around $8,0 * 10^8$ to $1,99 * 10^8$, i.e. about a factor of 4.

Indeed, for coarser grid levels level 1 and 2 the new layout decreases the performance drastically. Especially for the coarsest level the runtime multiples by a factor of 4. The coarse grid calculations are so fast, that the overhead for creating the new data structure has a big impact on the execution time. Additionally another effect has to be considered. When the grid consists of fewer elements the data vectors become much smaller. For grid Level 4 the data vectors contain nearly a million entries while there are only 3000 during grid level 1 calculations. In this case the jumps in the main memory caused by the ijk $\leftrightarrow$ npoi transformation are of a smaller extent and the chance is much higher that a value is needed which already resides in the cache. The total amount of needed data can be calculated with the formula:

$$Mem_{required} = Num_{vectors} * Length_{vectors} * sizeof(floattype) \quad (1)$$

where $Mem_{required}$ represents the total amount of data which is in use during the inner loop. $Num_{vectors}$ is the count of data vectors within the inner loop, while $Length_{vectors}$ determines the number of entries contained by the equal sized vectors. At last $sizeof(floattype)$ gives the size in KB of the uses floating point data format.

For all grids the number of needed vectors within the inner loop is 14. The datatype used within Fastest and sipsol is double precision. With these values the result is:

$$14 * 3000 * 8bytes \approx 0.3\,\text{MB}$$

As mentioned in section 2.1 the used AMD CPUs possess a Level 3 cache of 16 MB. That's why all vectors fit into cache. Therefore all data resides in L3 cache after some time during every iteration and the following calculations can be executed without additional memory operations. The same argumentation can be used for grid level 2 with its 18000 entries per data vector. The total amount of data is here around 2 MB and also fits in cache completely. This is not the case for grid level 4 since the total amount of needed data is here over 100 MB. The level 3 data is around 14 MB. Theoretically this can fit into the cache, but in practice some of the data will be removed and later be loaded again. So changes to the data structure will have an influence on level 3, but with a limited extend.

It must be stated, that these values are only valid for the used test case and the corresponding configuration files which determine the block size and partition of the grid. They determine the size of the specific vector size for the individual grid levels. So for bigger geometries it will be the case that even the smaller grid levels contain enough data, so that the new data layout will improve performance here as well.

5 FUTURE WORK

The main step will be to implement a parallelization based on threading. With the new data structure it is possible to parallelize over the lines within a diagonal. The drastically increased single core performance and the improved memory access behavior are the basis for an efficient, threading-based parallelization which is needed to take advantages of modern multi core CPUs. This parallelization has to take the length of the single diagonals into account to reach best performance.

On the other hand it should be investigated how it will influence the performance to use 2D grids with dynamic size in memory. In the current version, a static portion of memory is allocated for each diagonal independent of their size. Dynamic allocation would allow to save memory. In the case of dynamic grids their size is dependent on the longest line per diagonal. Figure 6 illustrates this fact. A drawback in that case is the loss of the possibility to use constant offsets when accessing neighboring diagonals or lines. These must then be calculated dynamically, too.

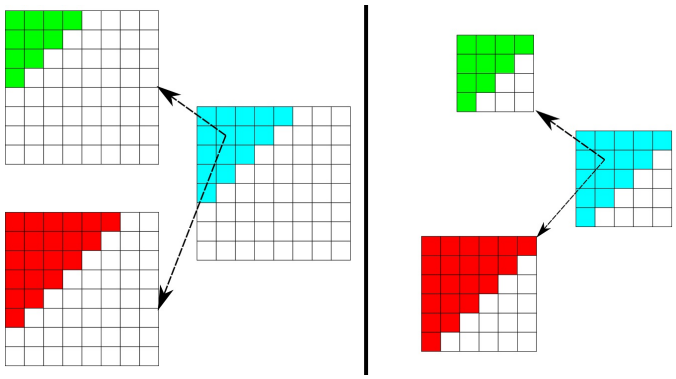

Figure 6: **Static and Dynamic Memory Fields in Comparison**: The left side shows the static allocation which is realized in the code. On the right side the dynamic case is depicted, which allows to save memory, but creates overhead for finding neighboring elements. Details in the text.

The last but very important point was touched several times during the paper. It must be investigated if it makes sense to port more work from the outer solver to the inner solver. Then there will be more successive iterations of the inner loop. So the optimizations in sipsol will have a greater impact on the overall performance. In the used version the only criteria to terminate the inner loop is to reach the maximum loop count. This maximum value is passed to sipsol at calling time and ranges only between 5 and 25. The higher this value the higher the effect of the here described code adaptions. Further investigations are needed to understand if such an increase of iterations would lead to faster or more exact solutions.

An additional approach would be the use of a dynamic abortion criteria which is based on a given target-precision or -improvement. So the number of iterations is decided in place. In that case this threshold determines the effect of the changes presented here.

As a last point it should be mentioned that also the structure of the v-cycles is an important point of interest. Because if one increases the number of runs on the finer grid levels one increases the role of the code modifications.

Only in the case of long iterations over fine grids the here presented data structure can play out its full potential and lead to significant speedups.

6 CONCLUSION

Our investigations have shown that the recent Fastest branch can be improved by employing a different memory access pattern within the "'Strongly Implicit Solver"' routine. The reasons for this problems have been exposed an a new data layout was implemented. This new version is able to nearly double the performance of the solver on a single core if the grid levels are fine i.e. the incoming vectors are too big to fit in processor cache. So the new developed version of the Strongly Implicit Solver features the ability for an efficient parallelization of the calculation, especially for large sized input vectors and the resulting larger 3D diagonals which are the elements of parallelization.

ACKNOWLEDGMENTS

We thank Doerte Sternel and Stefan Kneissl for the assistance in configuring and compiling Fastest and the discussions about the code. We also thank Christian Iwainsky for the assistance when getting the screenshots of Vampirtrace.

REFERENCES

[1] FNB: Project Site for Fastest / Technische Universtiaet Darmstadt. 2013 (1). – Forschungsbericht. – `http://www.fnb.tu-darmstadt.de/forschung_fnb/software_fnb/software_fnb.de.jsp`

[2] Patankar, Suhas V.: *Numerical Heat Transfer and Fluid Flow*. Hemisphere Publishing Corporation, New York, 1980

[3] Scheit, Christoph ; Hager, Georg ; Treibig, Jan ; Becker, Stefan ; Wellein, Gerhard: Optimization of FASTEST-3D for Modern Multicore Systems. In: *CoRR* abs/1303.4538 (2013)

[4] Kreutzer, Moritz ; Hager, Georg ; Wellein, Gerhard ; Bishop, Alan R.: A unified sparse matrix data format for modern processors with wide SIMD units. In: *arXiv* (2013)

[5] Leung, Shun tak ; Zahorjan, John: Optimizing Data Locality by Array Restructuring. 1995. – Forschungsbericht

[6] Stone, H. L.: Iterative Solution of Implicit Approximations of Multidimensional Partial Differential Equations. In: *SIAM J. Num. Anal.* 5 (1968), Nr. 3, S. 530–558