Optimized Stencil Computation Using In-Place Calculation on Modern Multicore Systems

Werner Augustin[1], Vincent Heuveline[2], and Jan-Philipp Weiss[1]

[1] SRG New Frontiers in High Performance Computing
[2] RG Numerical Simulation, Optimization, and High Performance Computing
Karlsruhe Institute of Technology, Universität Karlsruhe (TH), Germany
{werner.augustin,vincent.heuveline,jan-philipp.weiss}@kit.edu

Abstract. Numerical algorithms on parallel systems built upon modern multicore processors are facing two challenging obstacles that keep realistic applications from reaching the theoretically available compute performance. First, the parallelization on several system levels has to be exploited to the full extent. Second, provision of data to the compute cores needs to be adapted to the constraints of a hardware-controlled nested cache hierarchy with shared resources.

In this paper we analyze dedicated optimization techniques on modern multicore systems for stencil kernels on regular three-dimensional grids. We combine various methods like a compressed grid algorithm with finite shifts in each time step and loop skewing into an optimized parallel in-place stencil implementation of the three-dimensional Laplacian operator. In that context, memory requirements are reduced by a factor of approximately two while considerable performance gains are observed on modern Intel and AMD based multicore systems.

Keywords: Parallel algorithm, multicore processors, bandwidth-bound, in-place stencil, cache optimization, compressed grid, time blocking.

1 Introduction

The advent of multicore processors has brought the prospect of constantly growing compute capabilities at the price of ubiquitous parallelism. At least at a theoretical level, every new processor generation with shrinking die area and increasing core count is about to deliver the expected performance gain. However, it is well known that the main memory speed cannot keep up with the same rate of growth. Hence, the at all times existing memory gap becomes more and more pronounced. Hardware designers try to mitigate the gap by a hierarchy of nested and shared caches between the main memory and the highly capable compute cores. Moreover, sophisticated techniques and hardware protocols are developed in order to overcome existing bottlenecks. Hence, beside the issue of fine granular parallelism, the programmer has to consider the hierarchical and parallel memory design.

H. Sips, D. Epema, and H.-X. Lin (Eds.): Euro-Par 2009, LNCS 5704, pp. 772–784, 2009.

On modern multicore systems, on coprocessor devices like graphics processing units (GPUs), or on parallel clusters computation is overlapped with communication in order to optimize performance. However, many applications - especially numerical algorithms for the solution of partial differential equations - are bandwidth-bound in the sense that it takes more time to transfer data to the cores than it takes to process the same amount of data on the compute cores. The algorithms can be characterized by their computational intensity $I = f/w$ that is defined by the ratio of the number f of performed floating point operations (flop) per number w of transferred words (8 byte in double precision). For bandwidth-bound algorithms there is an upper bound for the effective performance P_{eff} given by the system bandwidth B (in byte/s) and the computational intensity I. This upper bound reads $P_{\text{eff}} \leq BI/8 = fB/(8w)$ for double precision and is often far less than the theoretically available compute performance. Basic routines of linear algebra like BLAS 1 or 2 routines have a computational intensity of order $O(1)$ with respect to the problem size N (i.e. vector size). Higher values can be for example achieved for lattice Boltzmann kernels (still O(1)) or FFT kernels ($\mathrm{O}(\log N)$). For BLAS 3 routines, the computational intensity is asymptotically growing linearly in N. For particle methods computational intensity is also increasing with problem size. One of the worst case routines is the `DAXPY` operation with $I = 2/3$. For the seven-point Laplacian stencil considered in this work we find $I = 4$.

Due to the restricting properties of the algorithms under consideration, a hardware-aware and memory hierarchy-adapted parallel implementation is crucial for best possible performance. On local-store-based architectures like the STI Cell processor or stream computing platforms like GPUs all memory transfers are managed explicitly by the user or semi-implicitly by the software environment. On modern multicore processors data provision is handled by hardware-controlled caches where the cache structure is hidden and transparent to the programmer. Moreover, cache memory is not addressable.

The goal of this paper is an efficient implementation of a 3D stencil application on modern cache-based multicore systems where the OpenMP programming approach is chosen for parallelization. A considerable performance increase can be achieved by applying an in-place implementation with a single array on a compressed grid. As an additional benefit, memory usage is reduced by a factor of approximately two. Typical multicore optimization techniques are applied as well in that context. This work is extending part of the contribution of [3] where dedicated techniques are developed in a similar context assuming out-of-place implementation with Pthreads. In the comprehensive work [2] in-place implementations are addressed without giving details on the algorithm, its implementation or any performance results. Our contribution shows the clear advantage of this algorithmic modification. For more details on possible applications of the developed techniques as well as existing schemes we refer e.g. to [4,6,7,8] and references therein.

This paper is organized as follows. Section 2 is dedicated to the framework of the considered stencil kernels. In Section 3 we provide a description of the

in-place stencil algorithm under consideration and describe the applied optimization techniques in more detail. Performance results are presented in Section 4 for different multicore systems ranging from Intel Clovertown and Nehalem, AMD Opteron, Barcelona and Shanghai, to Intel Itanium2. We conclude with a summary in Section 5.

2 Stencil Kernels

Stencil kernels are one of the most important routines applied in the context of structured grids [1] and arise in many scientific and technical disciplines. In image processing for example, stencils are applied for blurring and edge detection. More importantly, the discrete modeling of many physical processes in computational fluid dynamics (e.g. diffusion and heat transfer), mechanical engineering and physics (e.g. electromagnetism, wave theory, quantum mechanics) involves application of stencils. Stencils originate from the discretization of differential expressions in partial differential equations (PDEs) by means of finite element, finite volume, or finite difference methods. They are defined as fixed subset of nearest neighbors where the corresponding node values are used for computing weighted sums. The associated weights correspond to the coefficients of the PDEs that are assumed to be constant in our context. Stencil computations are endowed with a high spatial locality that expresses interactions restricted to nearest neighbors. Global coupling, as observed for the Laplacian/Poisson or the heat equation, is induced across several stencil iterations or by inversion of the associated matrix. In three dimensions 7-point (6+1), 9-point (8+1), or 27-point (6+8+12+1) stencils are common. The associated directions correspond to the faces, corners, and edges of a cube plus its center. Since the stencil only depends on the geometric location of the nodes, all necessary information can be deduced from the node indices. Due to the limited amount of operations per grid point with corresponding low computational intensity, stencil kernels are typically bandwidth-bound and have only a limited reuse of data. On most architectures stencil kernels are consequently running at a low fraction of theoretical peak performance.

In this paper, we consider a three-dimensional cubic grid with edge length N and a scalar value associated to each grid point. The grid is mapped to a one-dimensional array of size $n = N^3$ plus ghost layers. In our case we consider cyclic boundary conditions across the physical boundaries of the cubic domain. The 7-point Laplacian or heat equation stencil

$$U^{m+1}_{i,j,k} = \alpha U^m_{i,j,k} + \beta \left[U^m_{i+1,j,k} + U^m_{i-1,j,k} + U^m_{i,j+1,k} + U^m_{i,j-1,k} + U^m_{i,j,k+1} + U^m_{i,j,k-1} \right] \quad (1)$$

is accessing six neighbors with stride $\pm 1, \pm N$, and $\pm N^2$ in lexicographical order. These long strides require particular care to ensure data reuse since cache sizes are limited and much smaller than the problem size in general. In (1) $8n$ flop are performed on a vector of length n while at least $2n$ elements have to be transferred from main memory to the cores and back.

The considered Laplacian stencil scheme falls into the class of Jacobi iterations where all grid nodes can be updated independently and the order of the updates

does not matter. However, the updated nodes cannot be stored in the same array as the original values since some values are still needed to update other neighbors. For this reason so called out-of-place implementations with two different arrays for read and write operations are used. Parallelization of Jacobi-like methods by domain sub-structuring is straightforward since only ghost layers corresponding to the spatial and temporal width of the stencil need to be considered. For Gauss-Seidel-like stencil codes (e.g. for time-dependent problems) with interlaced time dependencies computation can be performed in-place with a single array for read and write operations, i.e. all updated values can be stored in the array of input values. However, the order of the grid traversal is decisive and the procedure is typically sequential. Multi-color approaches (red-black coloring) or wavefront techniques need to be considered for parallelization.

3 Description of the In-Place Stencil Kernel and Applied Optimization Techniques

Usually stencil operations are implemented as two-grid algorithms with two arrays where solutions of consecutive time steps are stored in alternating manner by swapping the arrays in each time step. As a consequence, data has to be read and written from and to different locations in main memory with significant delays due to compulsory cache misses caused by write operations. Considering the actual algorithm, these write cache misses are unnecessary. Therefore, attempts by using cache-bypassing mechanisms were made with varying success in [3] as well as in our study. Another possible remedy is to merge both arrays in an interleaved manner with some additional effort in the implementation [5]. But this procedure does not improve the additional memory requirements.

In order to tackle problems of memory consumption and pollution of memory transfers we are using an approach similar to the grid compression technique presented in [5] for lattice Boltzmann stencils. In this approach updated values are stored in-place and old input data is overwritten. Since grid point updates depend on all neighbors, a specified number of grid points has to be stored temporarily. This temporary storage within the same array results in a shift of the coordinates for the updated values. The underlying algorithm is detailed in the sequel.

3.1 One-Dimensional In-Place Computation

For a detailed description of the in-place stencil algorithm we start with the one-dimensional stencil operation

$$y[i] = \alpha x[i] + \beta(x[i-1] + x[i+1]). \tag{2}$$

Simple replacement of $y[i]$ by $x[i]$ yields unresolvable dependencies. But if the array is skewed with a shift of one there are no more dependencies considering

$$x[i-1] = \alpha x[i] + \beta(x[i-1] + x[i+1]). \tag{3}$$

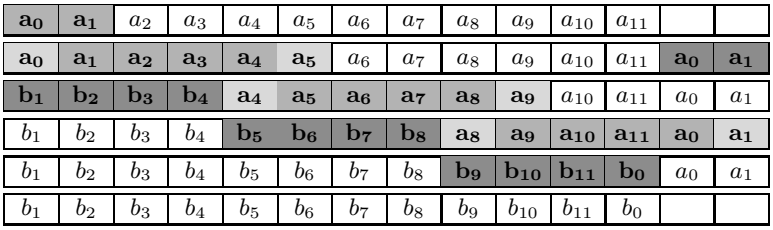


Fig. 1. One-dimensional, space-blocked, and shifted stencil operation

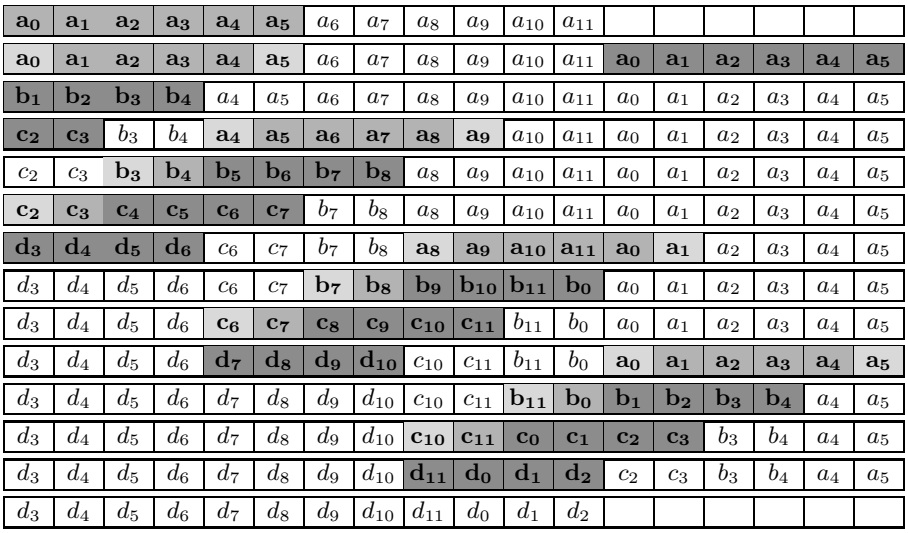


Fig. 2. One-dimensional, space-blocked, time-blocked and shifted stencil operation

With additional spatial blocking included the computation scheme is shown in Figure 1. The different rows represent snap-shots of the grid array after a blocked computation. Lower indices denote the grid coordinate in the problem domain. Letters a, b, c, etc., indicate the values of the original and subsequent time steps. The current changes are highlighted with dark gray background, the cells providing input for the next computation are highlighted with lighter gray. Additionally, cells at the block boundaries touched only once are marked with lightest gray. The first step consists of copying the left boundary to the right to allow for computation of cyclic boundary conditions. The consecutive steps show blocked and shifted computations. In the last row all updated values of the next time steps are available. The whole domain is shifted to the left by one.

In the next example in Figure 2 time-blocking is added. Here, three consecutive time steps are computed immediately within one block in order to improve data reuse and cache usage. Every block update still follows formula (3), i.e. every update uses data from itself and the two neighboring cells to the right,

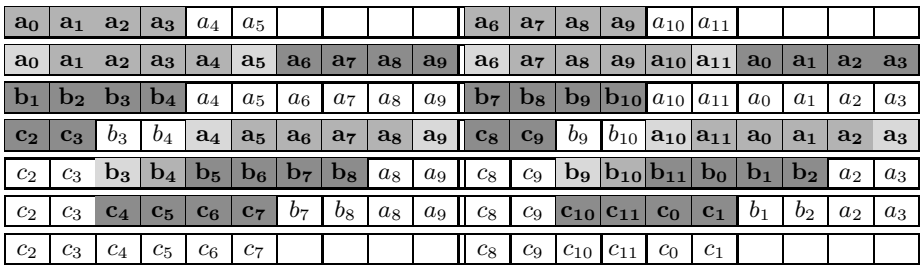


Fig. 3. One-dimensional, space-blocked, time-blocked, and shifted stencil operation on two cores

resulting in a skew of two between consecutive block computations (skew is one for out-of-place time skewing with two grids). Data flows from light gray to dark gray again. Light gray cells can be interpreted as cache misses because they are used but have not been computed in the previous block computation. As can be seen from the figure, cache misses are only slightly increased in case of temporal blocking. The final result is shifted to the left by the number of blocked time steps (by three in our case).

Our third example in Figure 3 shows the boundary treatment at the domain interfaces between different cores. The boundary overlap of the previous example has to be replicated for every core to allow parallel computation. This results in the fact that a number of boundary cells (equal to the number of blocked time steps) have to be calculated on both involved cores. This leads to some small computational overhead.

3.2 Multi-dimensional In-Place Computation

The results of the previous section can be applied to the two-dimensional case in a straightforward way. The corresponding computational pattern is shown in Figure 4. Different shades of gray denote the sequence of computation for the respective blocks.

The three-dimensional case is exemplified by means of a *C* code snippet in Figure 5 where initialization of appropriate boundaries is omitted. Parameters

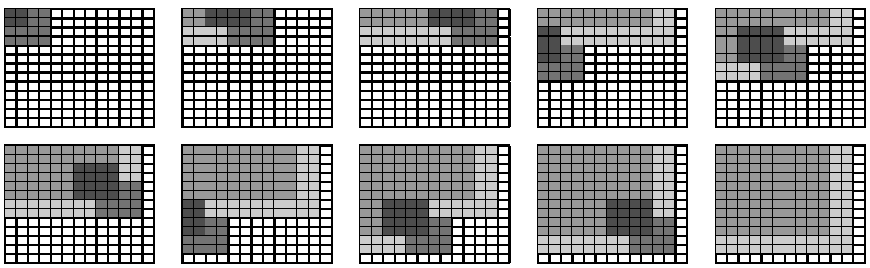


Fig. 4. Two-dimensional, space-blocked, time-blocked and shifted stencil operation

```
for( bz = 0; bz < zsize ; bz += bsz )
  for( by = 0; by < ysize; by += bsy )
    for( bx = 0; bx < xsize; bx += bsx )
      for( t = 0; t < tmax; t++) {
         tc = tmax - t - 1;
         xmin = max(0, bx - 2*t); xmax = min(xsize + 2*tc, bx+bsx - 2*t);
         ymin = max(0, by - 2*t); ymax = min(ysize + 2*tc, by+bsy - 2*t);
         zmin = max(0, bz - 2*t); zmax = min(zsize + 2*tc, bz+bsz - 2*t);
         for( z = zmin; z < zmax; z++)
           for( y = ymin; y < ymax; y++)
             for( x = xmin; x < xmax; x++)
                a[x, y, z] = stencil(a[x+1, y+1, z+1],
                   a[x  , y+1, z+1], a[x+2, y+1, z+1], a[x+1, y  , z+1],
                   a[x+1, y+2, z+1], a[x+1, y+1, z  ], a[x+1, y+1, z+1]);
      }
```

Fig. 5. *C* example code for the three-dimensional, space-blocked, time-blocked and shifted stencil operation

`xsize`, `ysize` and `zsize` denote the size of the computational domain. The size of the spatial block is specified by `bsx`, `bsy` and `bsz`. And finally, `tmax` is the number of blocked time steps.

3.3 Additional Optimization Techniques

In the following list applied multicore optimization techniques are given:

- **Parallel data allocation** is essential for performance on NUMA-architectures. Therefore, threads are pinned to specific cores and memory initialization is explicitly done by the same mapping.
- **Time and space blocking** is applied for reuse of (2nd level) cached data.
- **Cache-bypassing** is used to prevent unnecessary cache line loads due to write misses. Current processors access main memory in the granularity of cache lines (usually 128 bytes). Writing one double floating point value (8 bytes) therefore involves reading a complete cache line from memory, changing the corresponding bytes and writing the whole line back. Special SSE-instructions (in our context `mm_stream_pd`) allow to bypass the cache and write to memory directly, without wasting memory bandwidth for data which will be instantly overwritten anyway.
- **Parallelization** is done only along the z-axis. On a 512^3 grid with five blocked time steps an overhead of approximately 16% due to redundant computation at overlapping domains between processor cores is introduced (in the case of 8 threads). Parallelization along all axes would reduce this overhead to approximately 12%. With only three blocked time steps both values decrease to 9% and 7%.

Optimizations not considered in our implementation are listed below:

- Based on the results in [3] **SIMDization** was not considered to be valuable enough.
- **Register blocking** was briefly checked but is not used because no performance gain was observed (actually there is an implicit register blocking of two in x-direction when using the SSE-instruction for cache-bypassing).
- **Explicit software prefetching** is not used although a comparison with upper performance bounds in Section 4 suggests to investigate this topic.

4 Performance Results

In this section we present performance results on the platforms described in Section 4.1. Our goal is to evaluate and validate the proposed approach across different platforms. For all test runs we consider stencil computation on a 512^3 grid which requires at least 1 GB of memory.

The left diagram of each figure compares an out-of-place implementation (with and without cache bypassing) and our in-place algorithm in terms of grid updates. Three relevant vector operations (copy operation $y[i] = x[i]$ with and without cache bypassing and vector update $x[i] = x[i] + a$) are added for reference. These results should be interpreted as upper bounds representing the minimum necessary memory operations. Grid updates are chosen as unit of comparison in order to avoid confusion between single- and bi-directional memory transfer rates. The right diagram compares an out-of-place implementation with and without cache bypassing and our in-place algorithm in terms of GFlop/s. On top of the single time step results performance improvements for different time-blocking parameters are shown. A single grid node update counts 8 flop.

4.1 Test Environment: Hardware and Software

Our approach of an in-place stencil algorithm was tested and compared to other algorithms on a couple of cache-based multicore architectures:

- **2-way Intel Clovertown:** a 2-way SMP node with quad-core Intel Xeon X5355 processors running at 2.66 GHz clock frequency
- **2-way Intel Nehalem:** a 2-way SMP node with quad-core Intel Xeon E5520 processors running at 2.27 GHz clock frequency
- **2-way AMD dual-core Opteron:** an HP ProLiant DL145 G2 server, a 2-way SMP node with AMD dual-core Opteron 285 processors with 2.66 GHz clock frequency
- **4-way AMD dual-core Opteron:** an HP ProLiant DL585 server, a 4-way SMP node with AMD dual-core Opteron 8218 processors with 2.66 GHz clock frequency
- **4-way AMD quad-core Barcelona:** a 4-way SMP node with AMD quad-core Opteron 8350 (Barcelona) processors running at 2.0 GHz clock frequency

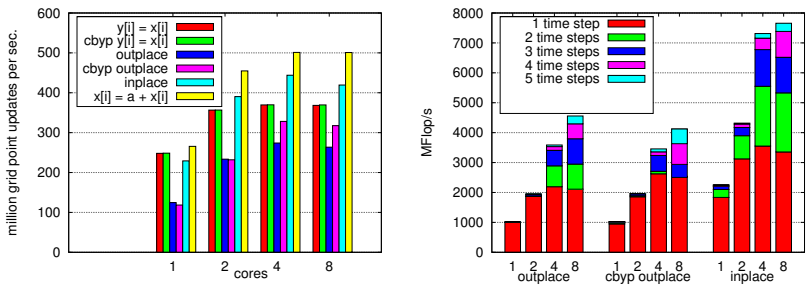
- **2-way AMD quad-core Shanghai:** a 2-way SMP node with AMD quad-core Opteron 2384 (Shanghai) processors running at 2.7 GHz clock frequency
- **8-way single core Intel Itanium2:** an HP Integrity RX8620 server, a 16-way single core Itanium2 Madison server partitioned in two 8-way nodes with 1.6 GHz clock frequency.

Beside the Intel Clovertown and Itanium2 (8 processors connected to two memory banks in interleave mode) all SMP systems are typical NUMA architectures with one memory bank per processor. All stencil algorithms are implemented in *C* using *OpenMP* as parallelization paradigm. They are compiled using an Intel C compiler version 11.0. On the platforms with Intel processors the optimization flag `-fast` is used. The code for the AMD processor based platforms is compiled with `-O3 -ipo -static -no-prec-div -xW`. On the Itanium2 platform we revert to the Intel C compiler version 10.1 because of severe performance problems. Presumably the compiler fails to get the software prefetching right resulting in 30-50% less performance.

4.2 Stencil Performance Across Several Platforms

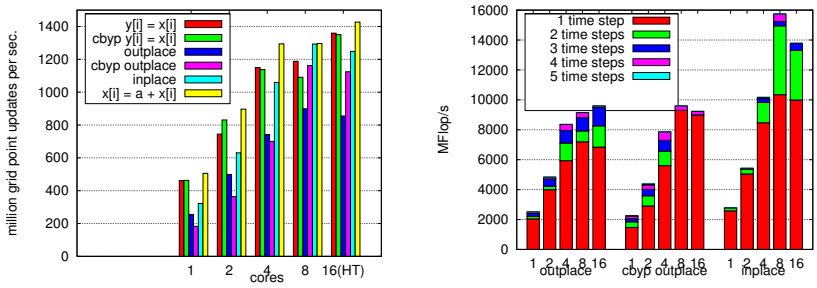
The Intel Clovertown results are shown in Figure 6. The saturation point of the memory bandwidth is almost reached with two cores. For four and eight cores the conventional out-of-place results agree very well with the ones presented in [3] and draw near the upper bound. As also mentioned in the article referenced above, adding cache-bypassing shows the same disappointingly low performance improvements. In-place computation improves these results by about 30%. The simpler memory access pattern also improves the effect of time-blocking considerably reaching an improvement of approximately 70%.

The results for the Intel Nehalem in Figure 7 show a performance boost of a factor of three compared to the Clovertown results. Here, the integrated memory controller brings considerable improvements for this bandwidth-bound kernel. The last column in each plot shows minor benefits or even a degradation from *Hyper-Threading* (HT) with two threads per core. There is no considerable profit



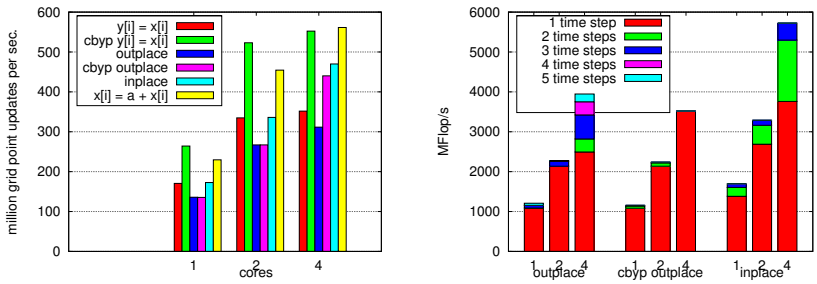
(a) Stencil performance **(b)** Algorithm comparison

Fig. 6. Intel Clovertown, quad-core, 2-way node, 512x512x512 grid



(a) Stencil performance **(b)** Algorithm comparison

Fig. 7. Intel Nehalem, quad-core, 2-way node, 512x512x512 grid



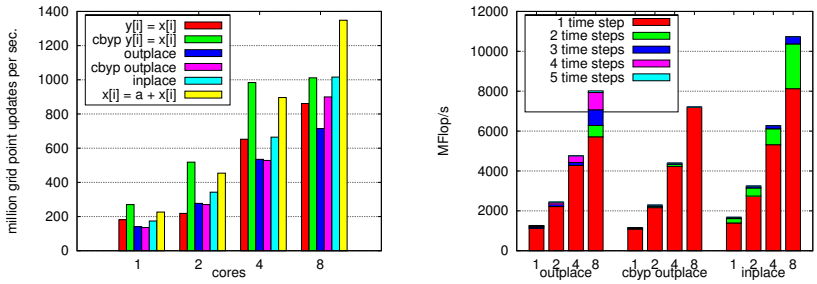
(a) Stencil performance **(b)** Algorithm comparison

Fig. 8. AMD Opteron, dual-core, 2-way node, 512x512x512 grid

from time-blocking in the out-of-place version of the algorithm. Cache-bypassing becomes effective when all cores are running. We find significant performance improvements when ramping up from two to eight cores.

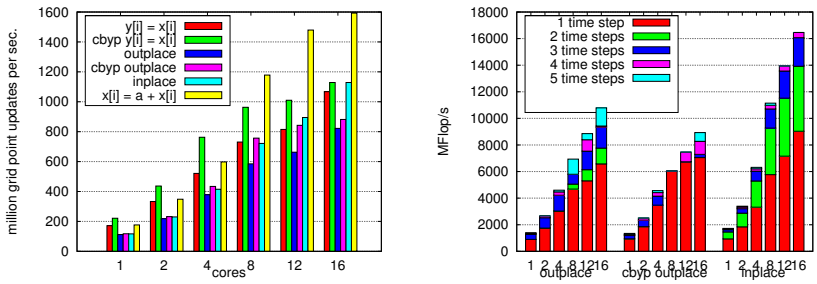
The results of the AMD dual-core processors are shown in Figure 8 and Figure 9. Results for the 4-way node are considerably better for the case of 4 cores because of the total of 4 memory banks instead of 2 for the 2-way node. The performance improvement between the out-of-place and in-place variants are at relatively low 7% for the version without time-blocking. This benefit increases to 40% for the 2-way nodes (13% and 27% for the 4-way nodes respectively) in case of time blocking.

The results of the AMD quad-core Barcelona processors are shown in Figure 10. While the results for the conventional out-of-place algorithm are similar to the ones presented in [3] (though one should be careful because of the hardware differences and the different clock speeds) the additional cache-bypassing shows very little improvement in our study. Possibly, non-optimal SSE instructions were used. The in-place results also lag behind the potential upper bound marked by the respective vector operations. Like on all the other processors



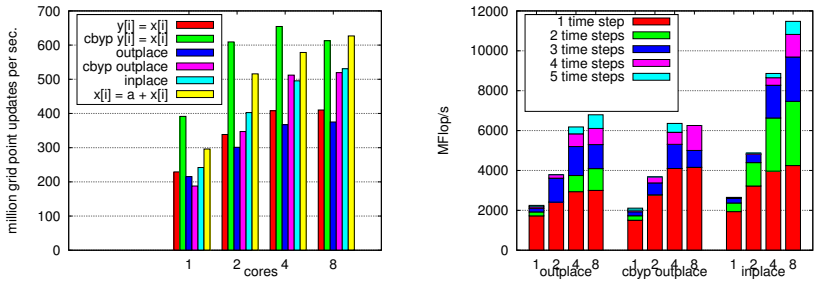
(a) Stencil performance **(b)** Algorithm comparison

Fig. 9. AMD Opteron, dual-core, 4-way node, 512x512x512 grid



(a) Stencil performance **(b)** Algorithm comparison

Fig. 10. AMD Barcelona, quad-core, 4-way node, 512x512x512 grid

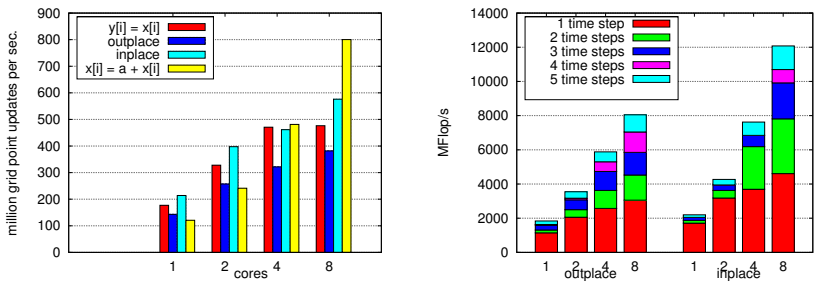


(a) Stencil performance **(b)** Algorithm comparison

Fig. 11. AMD Shanghai, quad-core, 2-way node, 512x512x512 grid

analyzed so far, time-blocking profits much stronger from the in-place algorithm than from the out-of-place version.

The results of the AMD Shanghai in Figure 11 clearly lag behind those of its competitor Intel Nehalem. Compared to the AMD Barcelona results we find a degradation by a factor of approximately 1.5 that is mainly attributed to the



(a) Stencil performance **(b)** Algorithm comparison

Fig. 12. Intel Itanium2, single-core, 8-way node, 512x512x512 grid

2-way and 4-way memory access. Performance values obtained by using cache-bypassing are very close to those from the in-place version.

Finally, the results of the Itanium2 architecture are presented in Figure 12. Due to the EPIC-architecture of the Itanium processor (EPIC = explicitly parallel instruction computing) performance heavily depends on compiler optimization which often conflicts with heuristics of memory and register resource allocation introduced by OpenMP. The compiler can be caused to disable essential techniques like software prefetching. This is a possible explanation why even the simple vector operations used for reference show bad performance. This observation is underlined by huge backdrops in performance for different versions of the Intel C compiler.

5 Conclusion

In this work we analyze the performance of an in-place implementation of a 3D stencil kernel on cache-based multicore systems using OpenMP. The proposed approach leads to considerable performance improvements especially considering combined time and space blocking techniques. In a future work the proposed scheme will serve as a building block for a larger Navier-Stokes solver on structured grids. Furthermore, an extension for Dirichlet and Neumann boundary conditions as opposed to the cyclic boundary conditions considered in this paper is currently under development.

Acknowledgements

The Shared Research Group 16-1 received financial support by the Concept for the Future of Karlsruhe Institute of Technology in the framework of the German Excellence Initiative and the industrial collaboration partner Hewlett-Packard.

We also thank the Steinbuch Centre for Computing (SCC) at Karlsruhe Institute of Technology (KIT) for its support and providing the computing nodes of

its clusters. The server with the AMD Barcelona cores was made available by the Lehrstuhl Informatik für Ingenieure und Naturwissenschaftler of the Department of Computer Science at KIT.

References

1. Asanovic, K., Bodik, R., Catanzaro, B., Gebis, J., Husbands, P., Keutzer, K., Patterson, D., Plishker, W., Shalf, J., Williams, S., Yelick, K.: The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley (December 2006)
2. Datta, K., Kamil, S., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Optimization and performance modeling of stencil computations on modern microprocessors. SIAM Review 51(1), 129–159 (2009)
3. Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., Yelick, K.: Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In: SC 2008: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, pp. 1–12. IEEE Press, Los Alamitos (2008)
4. Kamil, S., Datta, K., Williams, S., Oliker, L., Shalf, J., Yelick, K.: Implicit and explicit optimizations for stencil computations. In: MSPC 2006: Proceedings of the 2006 workshop on memory system performance and correctness, pp. 51–60. ACM Press, New York (2006)
5. Th. Pohl, M., Kowarschik, J., Wilke, K.: Iglberger, and U. Rüde. Optimization and profiling of the cache performance of parallel lattice Boltzmann codes. Parallel Processing Letters 13(4) (December 2003)
6. Vuduc, R., Demmel, J., Yelick, K.: OSKI: A library of automatically tuned sparse matrix kernels. Journal of Physics: Conference Series 16, 521–530 (2005)
7. Whaley, R., Petitet, A., Dongarra, J.: Automated empirical optimizations of software and the ATLAS project. Parallel Computing 27 (2001)
8. Wonnacott, D.: Time skewing for parallel computers. In: Proc. 12th Workshop on Languages and Compilers for Parallel Computing, pp. 477–480. Springer, Heidelberg (1999)