Multigrid and Gauss-Seidel Smoothers Revisited: Parallelization on Chip Multiprocessors

Dan Wallin, Henrik Löf, Erik Hagersten and Sverker Holmgren
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, SWEDEN
{dan.wallin, henrik.lof, erik.hagersten, sverker.holmgren}@it.uu.se

Abstract

Efficient solutions of partial differential equations require a match between the algorithm and the underlying architecture. The new chip-multiprocessors, CMPs (a.k.a. multicore), feature low intra-chip communication cost and smaller per-thread caches compared to previous systems. From an algorithmic point of view this means that data locality issues become more important than communication overheads. This may require re-evaluation of many existing algorithms.

We have investigated parallel implementations of multigrid methods using a temporally blocked, naturally ordered, smoother implementation. Compared with the standard multigrid solution based on the two-color red-black algorithm, we improve the data locality often as much as ten times, while our use of a fine-grained locking scheme keeps the parallel efficiency high.

While our algorithm initially was inspired by CMPs, it was surprising to see our OpenMP multigrid implementation run up to 40 percent faster than the standard red-black algorithm on an 8-way SMP system. Studying the smoother part of the algorithm in isolation often shows it performing two iterations at the same time as a single iteration with an ordinary red-black smoother. Running our smoother on a 32-thread UltraSPARC T1 (Niagara) SMT/CMP and a simulated 32-way CMP demonstrates the communication cost of our algorithm to be low on such architectures.

1 Introduction

Whenever there is a paradigm shift in computer architecture, algorithms for high performance computing need to be re-evaluated and possibly modified to assure that they still achieve the best possible performance. Recent development in microprocessor technology has been focused on simultaneous multi-threading (SMT) [6, 8] and chip-multiprocessing (CMP) [2, 8] architectures. All major processor vendors are at the moment releasing chip multiprocessors with several threads [9, 10, 11, 13]. The introduction of such processors has several implications for the design of efficient parallel algorithms. The per-thread cache size will be significantly smaller compared to SMPs built from many single-threaded CPUs. Running large input sets may therefore require optimizations for data locality. Further, minimizing communication and synchronization may not be of imperative importance any more, since communication between the threads can be performed on-chip via fast shared caches. In the light of these architectural changes, we re-consider the parallelization of a widespread and important computational kernel in this paper.

The basic problem studied below is the solution of a large, structured system of equations using a multigrid scheme, see e.g. Wesseling [18]. This computation is found at the heart of scientific computing applications in a range of disciplines. Parallel and high-performance implementations have been studied by numerous authors during more than twenty years. In the algorithms, a basic *smoothing* operation is recursively applied to a sequence of array data structures. Standard smoothers such as Gauss-Seidel, SOR and ILU all have the same type of internal data dependencies, and they are traditionally parallelized using *multicolor* orderings of the grid points [1]. Unfortunately, the algorithms arising from these orderings are difficult to parallelize if cache-blocking techniques are added to increase data reuse. As an alternative, we propose to use a much more synchronization-intensive dataflow parallelization technique for the standard, natural ordering of the unknowns. This type of scheme enables us to exploit recently cached data in a better way than the standard multicolor technique.

Furthermore, the temporal blocking allows us to apply a sequence of smoothing operations at a cost that is only slightly larger than for applying the smoother once. This leads to reduced computational time for the multigrid scheme, since the number of multigrid iterations is significantly reduced while the cost per iteration is only marginally increased.

A dataflow parallelization technique for structured grid Gauss-Seidel-type smoothers was considered already in the very early days of parallel computing [14]. However, it was almost immediately abandoned since the multicolor schemes proved to be much more efficient on parallel systems where synchronization is expensive. The idea of utilizing temporal blocking to improve cache utilization in serial implementations of multigrid smoothers has also been considered earlier, both for the two-color red-black [17] and natural [15] orderings. The main contribution of this paper is to combine the two techniques, and to show that the resulting parallel multigrid scheme is faster than the standard method on CMPs.

The rest of this paper is structured as follows: First, we introduce the Gauss-Seidel smoother and the concept of grid orderings by considering the model problem solved in the experiments later. After this, we describe the parallel temporally blocked Gauss-Seidel smoother, and we evaluate this algorithm in detail in terms of performance and scalability. We continue by introducing multigrid methods, which we use to solve the model problem. Finally, we evaluate the performance of the multigrid solver. The experiments are performed on both traditional SMP systems and on CMPs. The results on the SMP system show our new smoother algorithm to perform two iterations at the same time as a single iteration with as standard red-black implementation. The reason for this is that the cache miss ratios are reduced by order of magnitudes at the expense of higher communication overhead compared to the red-black implementation. The results from running the codes on a CMP processor, show that the additional communication in the temporally blocked Gauss-Seidel kernel does not degrade the performance on such systems. This is caused by all communication taking place via a shared second level cache.

2 Smoothers for Structured Grids

For wide classes of application problems, a multigrid iterative method is the most efficient technique for solving systems of equations

$$Lu = f, \tag{1}$$

where u is the numerical solution to a partial differential equation (PDE), f is the discretized forcing function, and L is a stencil operator representing a discretization of the derivatives in the PDE on a structured grid. For a d-dimensional problem solved on a grid with N^d grid points, grid functions like u and f are stored in d-dimensional arrays with N^d entries, and data parallel stencil operators such as L are applied by computing weighted averages of function values at a set of neighboring grid points. The basic building block of a multigrid method is a *relaxation*, or *smoothing* iteration as described by Algorithm 1. In this section, we review the classical

Algorithm 1 Relaxation(u,f)

$u = 0$ {Initial guess}
while u not sufficiently improved **do**
  $u = S(u, f)$ {Apply smoother}
end while

setting where Algorithm 1 is used for solving Equation 1 on a single grid. The discussion of multigrid schemes, where the smoother is employed on a hierarchy of grids, is deferred to Section 9.

The standard smoothing operators S in Algorithm 1 are given by the Jacobi, Gauss-Seidel and SOR splittings, see e.g. Young [19]. For all these three methods, S is similar to L in the sense that it is applied by combining grid function values at a set of neighboring grid points. However, for the two latter methods, S is also different from L in the sense that it is not data parallel, the computations at the different grid points must be performed in a certain order.

As for all iterative methods, the computational time for Algorithm 1 depends on the time needed for a single application of the smoother and the total smoothing steps required. A major problem with the classical splitting methods mentioned above is that the number of steps grows with N, and in practice it is normally not feasible to compute an accurate solution u to the PDE using these schemes. The convergence is accelerated dramatically if the smoother is employed within a multigrid method.

We now introduce the PDE that will be solved in the experiments later, and we use this setting to further discuss Gauss-Seidel smoothers and the concept of grid point orderings. We consider a standard model problem, the Poisson equation with homogenous boundary conditions solved in the unit cube,

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega = [0,1]^3 \\ u &= 0 \quad \text{on } \partial\Omega. \end{aligned} \tag{2}$$

Using the standard 7-point stencil finite-difference discretization, the difference operator L is defined by

$$\begin{aligned} Lu_{(i,j,k)} \equiv \tfrac{1}{h^2}\Big(&6u_{(i,j,k)} - u_{(i+1,j,k)} - u_{(i-1,j,k)} \\ &-u_{(i,j+1,k)} - u_{(i,j-1,k)} - u_{(i,j,k+1)} - u_{(i,j,k-1)}\Big) = f_{(i,j,k)}. \end{aligned} \tag{3}$$

Here h is the step length of the uniform grid and the indices (i, j, k) run over all N^3 interior points in the grid. The relation (3) is equivalent to a symmetric, positive definite system of N^3 equations. A Gauss-Seidel smoother for this problem is easily implemented using the *in situ* computations:

$$\begin{aligned} u_{(i,j,k)} = Su_{(i,j,k)} \equiv \tfrac{1}{6}\Big(&u_{(i+1,j,k)} + u_{(i-1,j,k)} + u_{(i,j+1,k)} \\ &+u_{(i,j-1,k)} + u_{(i,j,k+1)} + u_{(i,j,k-1)} + h^2 f_{(i,j,k)}\Big). \end{aligned} \tag{4}$$

Here, the ordering of the computations determines if the right hand side terms $u_{(i,j,k)}$ of Equation 4 contain the old solution value or the already computed new value. Thus, Equation 4 represents a *family* of Gauss-Seidel smoothers, which produce different results for different orderings of the computations. Conversely, a given ordering corresponds to a specific set of data dependencies for the computation of the new values in u.

The most straightforward way to order the computations in (4) is the *natural* or *lexicographical* ordering. This corresponds to performing the computations using a simple triple loop over the indices (i, j, k), which leads to inherently sequential dependencies. Another popular way of ordering the computations is to use *multicolor orderings*. A special case of this ordering technique is the two-color red-black ordering. Here, odd and even points are updated in two separate sweeps. In this case the data dependency in

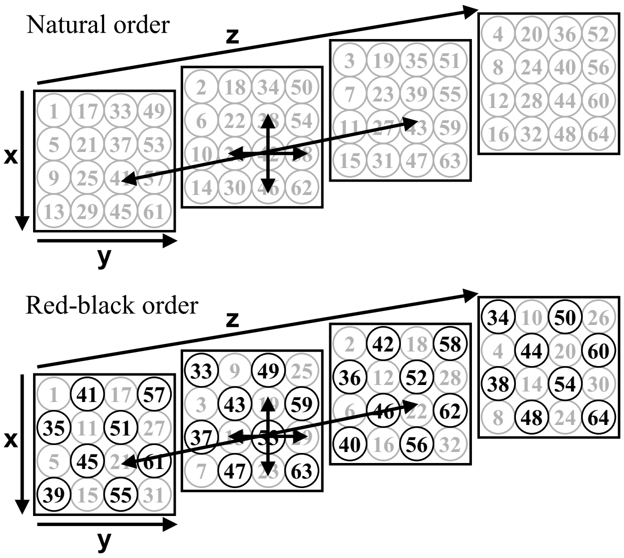
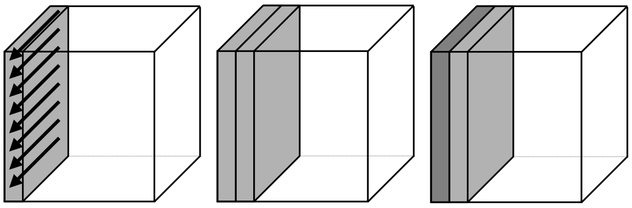
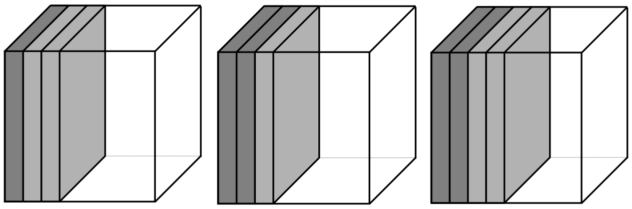


Figure 1. Natural and Red-black orders in a 3D problem. The 7-point stencil accesses data in all three directions.

the algorithm only dictates that the two sweeps must be performed in sequence, while each sweep is fully data parallel. An example of the two orderings and the 7-point stencil can be found in Figure 1.



(a) Apply the stencil to the first slice the first time (b) Apply the stencil to the second slice the first time (c) Apply the stencil to the first slice the second time



(d) Apply the stencil to the third slice the first time (e) Apply the stencil to the second slice the second time (f) Apply the stencil to the fourth slice the first time

Figure 2. Principle of temporally blocked Gauss-Seidel

3 Cache Optimized Smoothers

The natural Gauss-Seidel ordering utilizes the cache better than the red-black version as long as the data values are stored in sequential order and the cache is not large enough to store the entire problem. This is the consequence of the red-black algorithm traversing the data values in two separate sweeps, thus causing the data to be evicted from the cache before the second sweep is performed.

Since the solution of the problem requires the stencil to be applied many consecutive times, it is possible to further improve the cache performance. This can be done by reusing the data values before discarding them in the cache according to Sellappa et al. [15]. This technique is called the temporally blocked Gauss-Seidel (TBGS). The order of applying the stencil to the data values is shown in Figure 2 and works as follow: First the stencil is applied to all data points in slice 1 according to the figure. The arrows indicate that the stencil traverses all data points from the top row to the bottom row of the slice. Second the stencil is applied to all data points in slice 2, third the stencil is applied to all data points in slice 1 for a second time and fourth the stencil is applied to all data points in slice 3 the first time. The scheme continues like this until the stencil has been applied to all data points two times. This scheme is more cache efficient than the natural Gauss-Seidel ordering, since the stencil can be applied several times to each data point before it is being discarded from the cache. Mathematically, the temporally blocked Gauss-Seidel scheme is identical to the natural ordered Gauss-Seidel.

We define the number of times we traverse the data points from one corner to the diagonal corner as the number of *sweeps*. During a single sweep, the stencil can be applied several times to each data point, as we saw in the example above. The number of times the stencil is applied is called the number of *steps*. An important performance measurement therefore becomes the number of *steps* per *sweep*, called σ. In the previous example found in Figure 2 $\sigma = 2$. The natural Gauss-Seidel ordering has $\sigma = 1$. For the red-black ordering, $\sigma = 0.5$, since two sweeps, one for the data of each color, are required for the stencil to be applied to all data points.

The red-black ordering can also be optimized for locality in a similar way as for the natural Gauss-Seidel ordering. A thorough investigation of such techniques was performed by Weiss et al. [17]. These techniques include loop fusion, where the stencil is applied to the red and black data points within a single sweep ($\sigma = 1$), and loop blocking, where several steps are made in one sweep ($\sigma > 1$). These techniques manage very well to reduce the number of cache misses, since the data is reduced several times before eviction. We have found that the performance of such cache optimized red-black smoothers is equal to the performance of the temporally blocked Gauss-Seidel smoother. However, there is a drawback with the cache optimized red-black ordering: it is not easily parallelized since it introduces dependencies between the data points. The main reason for introducing the red-black ordering instead of the natural ordering was to decouple the dependencies.

4 Parallel Temporally Blocked Gauss-Seidel

A major contribution of this article is to show that the temporally blocked Gauss-Seidel scheme can be efficiently parallelized on shared-memory multiprocessors. The natural Gauss-Seidel ordering is seldom used in parallel cases since it requires fine-grained locking [7]. The locking leads to a high communication cost, which causes a poor speedup. However, we show that this communication overhead can be compensated by a better cache usage.

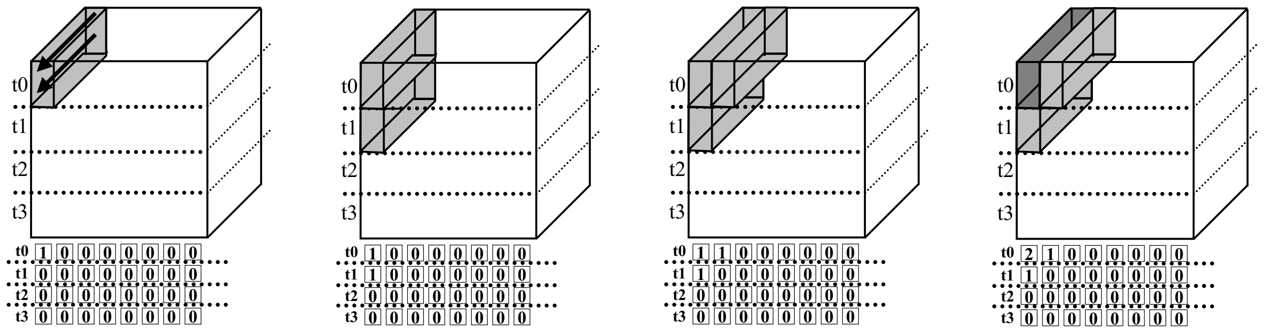
The proposed parallelization uses a flag-vector of size $t \times N$, where t is the number of threads and N is the problem size. The flags indicate how many times the stencil has been applied to the data elements of the corresponding slice according to Figure 3. The total size required for the flag-vector is negligible compared to the total N^3 problem size for realistic values of t and N.

A thread t can start executing on the next slice in step s if two conditions are fulfilled. The first condition (1) is that the thread $t-1$ already has performed step s. This condition prevents thread t from performing step s before thread $t-1$. The second condition (2) is that the flags for thread t and $t+1$ indicate that the same number of steps has been made. This condition prevents thread t to perform step $s+1$ before thread $t+1$ performs step s. These two conditions guarantee that the parallel execution yields the same results as the sequential execution.

The parallelization of the temporally blocked Gauss-Seidel smoother leads to a certain start time overhead. The startup overhead is caused by threads $t > 0$ having to wait for the thread above to complete its part of the slice. The behavior is similar to that of pipelining and the size of the overhead is $(t-1)/(\sigma N)$. In most cases, the cost is small since the problem size, N, normally is larger than the number of threads, t. The overhead is proportional to the number of sweeps. Since, σ indicates the number of steps per sweep, a large value of σ leads to a smaller start time overhead for a fixed number of steps.

5 Experimental Setup

In this article, most experiments have been run on an UltraSPARC IV+ shared-memory system. The baseline system consists of a 8-way processor card installed in a Sun server. The processor card has four CPU chips, each with two 1.5 GHz processor cores. The cores have separate 64 KB level one data caches. The two cores of each chip share an internal 2 MB level two cache and an external 32 MB level three cache. This system is from an architectural point of view a mixed 8-way CMP/SMP system.



(a) Perform step 1 on the first slice on t0.

(b) Perform step 1 on the first slice on t1.

(c) Perform step 1 on the second slice on t0.

(d) Perform step 2 on the first slice on t0.

Figure 3. Principle of parallelization of 3D temporally blocked Gauss-Seidel.

The UltraSPARC IV+ system is the computer with the best overall floating-point performance that is available to us at the moment. However, since this system is not a pure CMP-design and we want to evaluate the effect of the proposed algorithms on such systems, we have performed additional experiments on CMPs in Section 8.

The UltraSPARC IV+ system runs the Solaris 9 operating system. All codes were written in Fortran 90 and compiled with high-level optimization (*fast*) using the Sun Forte 8.1 Fortran 95 OpenMP compiler. We used double precision for all floating-point values which correspond to 8 B of data. The compilation was optimized for the processor and cache hierarchy of the system. We have chosen to disable the compiler-assisted software-prefetch mechanism. Since the success of the prefetching varies between different versions of the code the results were sometimes bumpy and hard to interpret. Our experiments from running with the prefetch enabled ensure us the overall results presented in the article are not obscured by this measure.

All runs were performed four times on a non-loaded system. We only report the results of the fastest of the four runs. The results normally varied less than one percent between the runs.

6 Sequential Performance

In this Section we evaluate the behavior of the Gauss-Seidel smoothers. The problem is run with three different problem sizes, where N equals 129 (2^7+1), 257 (2^8+1) and 513 (2^9+1). The total data size required for storage of the solution, u is 16.4 MB, 129 MB and 1030 MB respectively. Note that, the right hand side, f, of Equation 3 also requires the same amount of storage.

We start with the analysis of cache behavior and execution time of the sequential TBGS-smoother. The sequential execution time per step is presented in Table 1. The values indicate how long each step takes depending on how many steps are performed in each sweep, σ. We performed a total of 16 steps for each experiment. The results show that the execution time per step is larger for the RBGS-smoother than the baseline TBGS-smoother with $\sigma = 1$ for all problem sizes. However, the difference in execution time per step is much smaller for the $N = 129$ problem size. The reason is that at this problem size, the entire problem almost fits into the third level cache of the UltraSPARC IV+ processor. For the larger problem sizes, $N = 257$ and $N = 513$, two steps with the TBGS-smoother take less time than a single RBGS-smoother step thanks to the better cache usage in the second and third level caches. The cost of performing additional steps per sweep is low for the TBGS-

	RBGS	**TBGS**				
σ	**0.5**	**1**	**2**	**4**	**8**	**16**
N=129	0.094	0.086	0.071	0.061	0.070	0.074
N=257	2.488	1.508	1.048	0.814	0.698	0.641
N=513	19.95	12.12	8.975	7.429	9.975	12.58

Table 1. Execution time per step for the RBGS- and TBGS-solvers at different problem sizes on the UltraSPARC VI+ processor. σ is varied for the TBGS-solver.

smoother.

The TBGS-smoother operates on a number of slices at a given time. The size of each slice is the size of each data value times the size in the x- and y-directions, $8N^2$ B. The larger the number of steps per sweep is, the larger the number of slices needed to cache the data becomes. If $\sigma = 4$, step 1 is performed on slice z, step 2 on slice $z-1$, step 3 on slice $z-2$ and step 4 on slice $z-3$. Thus, the required number of slices is four. However, we also access data in slices $z+1$ and $z-4$ since the 7-point stencil is extended in these directions. We call the slices that are accessed in the computations the *active region*. Since the solution of the problem also has a right hand side, f, one additional slice of data is required for each σ. The cache size needed for the computation of each active region is equal to $2\sigma+2$. In Table 2, the required data size for each active region has been calculated.

σ	**1**	**2**	**4**	**8**	**16**
slices	4	6	10	18	34
N=129	0.5 MB	0.76 MB	1.3 MB	2.3 MB	4.3 MB
N=257	2.0 MB	3.0 MB	5.0 MB	9.1 MB	17 MB
N=513	8.0 MB	12 MB	20 MB	36 MB	68 MB

Table 2. Required data size for each active region.

The RBGS-smoother needs to fit the entire problem into cache to exploit temporal locality between steps. This is not the case for the TBGS-smoother since it is enough to fit the the active region in cache for an efficient cache usage. The number of cache misses will be reduced by a factor σ because of the temporal blocking as long as the active region fits in the cache. Hence, the execution time per step should decrease for larger values of σ. However, Table 1 shows that the execution time per step actually increases when $\sigma > 4$ for the $N = 129$ and $N = 513$ problem sizes. This can be explained by the fact that the active region increases in size when σ grows and it will eventually be too large to fit in some level of the UltraSPARC IV+ cache hierarchy. The level two cache is 2 MB and can store the active region at $\sigma \leq 4$ for the $N = 129$ problem size. Similarly, the third level cache size is 32 MB and is large enough to store all active regions for all problem sizes except at $\sigma > 4$ for $N = 513$.

To further increase the understanding of the sequential performance, the RBGS- and TBGS-smoothers have been analyzed using the StatCache tool developed by Berg et al. [3]. The tool uses a probabilistic approach that makes it possible to compute the cache miss ratios of any cache size from sparse data samples collected during a single run[1].

The cache miss ratio curves for the RBGS- and TBGS-smoothers generated by StatCache is presented in Figure 4. The problem size is $N = 129$ and the results assume a cache block size of 64 B. There are many interesting conclusions to draw from this figure: The cache miss ratio is almost proportional to σ for cache sizes between 4 and 16 MB. The RBGS miss ratios are almost twice as high as for TBGS1 (TBGS with $\sigma = 1$) for all problem sizes except for the case where the entire problem can be cached. This difference in performance can be explained by the fact that the RBGS-smoother cannot reuse data between steps since data from the first sweep have been evicted. The TBGS-smoother performs equally well for all values of σ at cache sizes up to about 512 KB. At larger problem sizes, the smoothers differ. As can be seen in Figure 4, the TBGS16-smoother has the lowest miss ratios for cache sizes larger than 2 MB. Between 512 KB and 4 MB we see that the cache miss ratios differ between TBGS1 and TBGS16. The TBGS1 cache miss ratio is still the half of RBGS. TBGS2 has the lowest ratio at 1 MB, TBGS4 at 2 MB and TBGS16 at 16 MB. The reason for these drops is that the active region requires a larger cache size for higher values of σ. The StatCache figures clearly show that for larger cache sizes, the miss ratio is halved for each doubling of σ for the TBGS-smoother.

The StatCache results for the $N = 257$ and $N = 513$ problem sizes are not presented here. However, they are very similar to the results for the $N = 129$ problem size.

7 Parallel Performance

We continue the article with a study of the parallel performance of the RBGS- and TBGS-smoothers. The TBGS-smoother was parallelized according to Section 4. The execution time per step is presented in Figure 5 for all problem sizes. The number of threads is varied between 1 and 8 in this experiment.

Adding several threads leads to a larger effective cache size for the studied architecture. The level two and level three caches of the UltraSPARC IV+ processor are shared between two cores. This means that if two threads run on the same processor, the effective level two and three cache size is unchanged. In the four-threaded runs the cache size is doubled and in the eight-threaded runs the size is quadrupled. However, the first level caches are private and doubles with each doubling of the number of cores.

In the previous section, we concluded that for all sequential runs the execution time per step is smaller for the TBGS-smoother than the RBGS-smoother. In the parallel cases this is only true for the runs with problem sizes larger than $N = 129$. There are two reasons for this behavior. The first reason is that in the TBGS-smoother, the ratio between communication and computation is larger the smaller the problem is since fewer computations are made before synchronization has to be made with the next thread. Since the computations are dependent on each other in the TBGS-smoother and are restricted by flag synchronization, the cost is much higher than in the RBGS-smoother. The second reason is that the entire problem fits in the third level cache for the $N = 129$ problem size. The major advantage with the TBGS-smoother is that the cache miss ratio is much lower than for the RBGS-smoother. In modern processors the access time to memory can be several hundreds of cycles. It is therefore crucial to avoid memory references for good performance. The additional cost of accessing the third level cache instead of the second level cache is not as severe.

Another interesting behavior in the different TBGS-smoothers can be observed in the execution time per step as the number of threads increases. As described in the previous section, the fastest sequential execution time per step is found for $\sigma = 4$ at problem sizes $N = 129$ and $N = 513$ because of the cache sizes of the computer. The same behavior can be observed for two threads. This is because the second and third level caches are shared and the cache size is unchanged for two threads. When more than two threads are used, the fastest execution time per step is instead found in TBGS with $\sigma = 8$ and $\sigma = 16$ because more cache space is available.

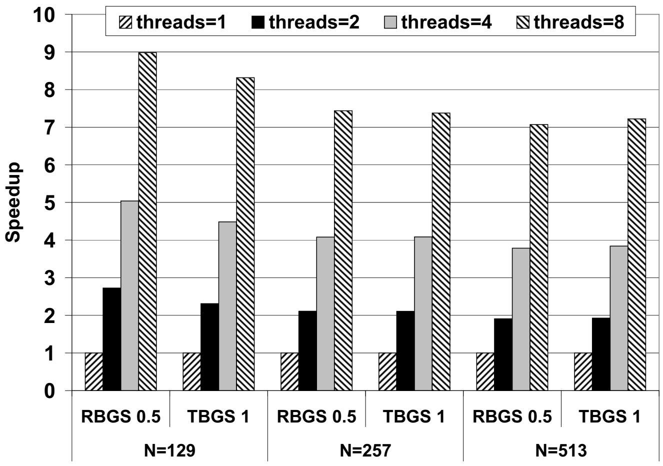


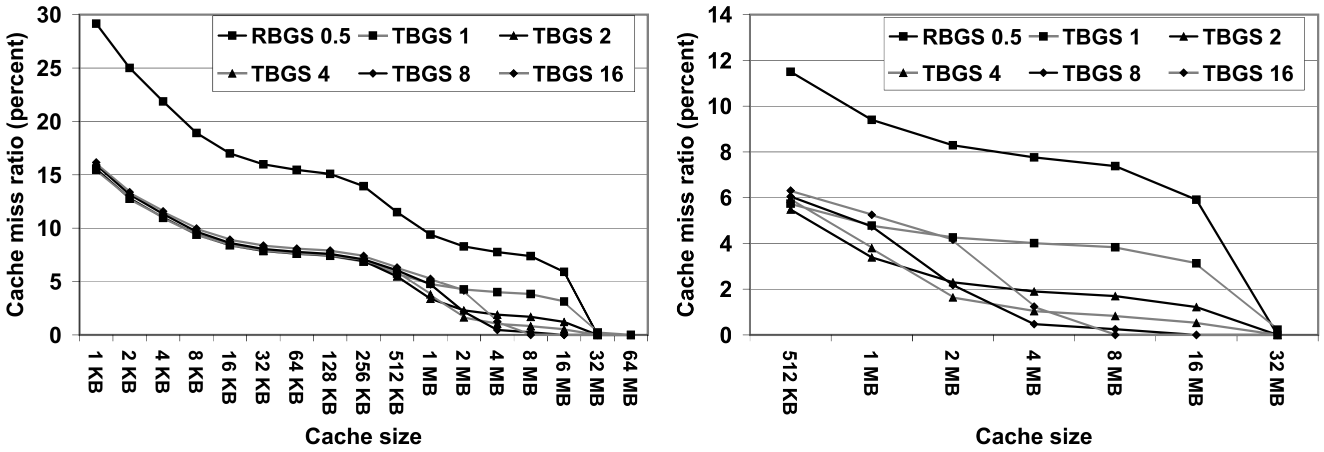
Figure 6. Speedup of the RBGS- and TBGS-smoothers for three different problem sizes on the UltraSPARC VI+ system.

Figure 6 shows that compared to the base-line sequential RBGS-smoother, the speedup on eight threads is between 7.1 and 9.0 for the parallel RBGS-smoother on the three problem sizes. The speedup is largest for the smallest problem size because the entire problem fits into the third level caches of the processors. The eight-thread TBGS-smoother shows a speedup of between 7.2 and 8.3 compared to the sequential TBGS-smoother depending on the problem size.

8 Scalability

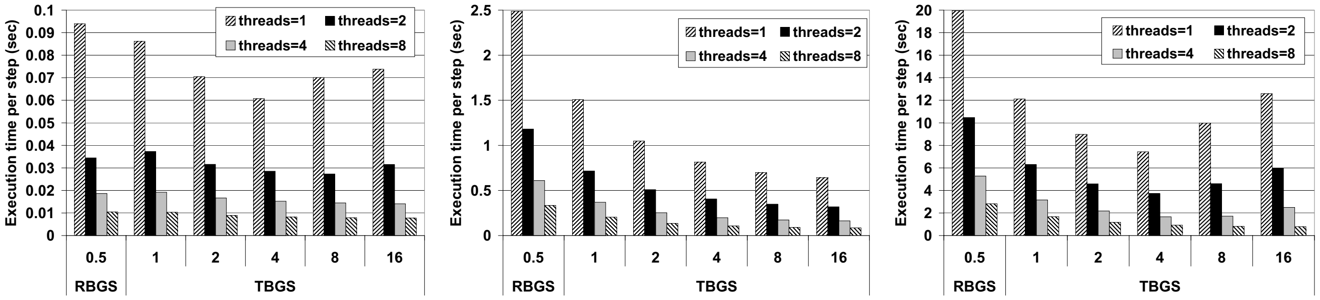
In this section, we will investigate the scalability of the TBGS- and RBGS-smoothers on present and future computer systems. We have therefore performed scalability experiments on a number of different architectures. The three studied systems are (1) a traditional 32-way cc-NUMA, (2) a 32-way SMT/CMP UltraSPARC T1 and (3) a simulated 32-way pure CMP. We have chosen not to include the previously studied UltraSPARC IV+ system in this section because it only can execute eight parallel threads.

[1] While StatCache's assumption is fully associative caches with random replacement, we regard these results as roughly representative for any fairly-associative cache.



(a) Cache miss ratios for cache sizes between 1 KB and 64 MB. (b) Cache miss ratios for cache sizes between 512 KB and 32 MB.

Figure 4. StatCache generated cache miss ratios at different cache sizes for the RBGS- and TBGS-smoothers at N=129.



(a) N = 129 (b) N = 257 (c) N = 513

Figure 5. Execution time per step for 1, 2, 4 and 8 threads on the UltraSPARC VI+ system.

The traditional cc-NUMA (1) consists of 32-processor UltraSPARC IIIcu based Sun E15K server. The server is a NUMA system with four processors per node. Each UltraSPARC IIIcu runs at 900 MHz and has a 64 KB first level data cache and a 8 MB second level cache. The data were allocated in parallel to avoid NUMA allocation effects.

The UltraSPARC T1 [8] processor can run 32 parallel threads and is built from eight four-threaded cores, each with an 8 KB level one data cache and a shared 3 MB level two cache. Since each core has four separate threads, which share the same pipeline, the processor is a eight-core 4-way SMT multiprocessor. The UltraSPARC T1 processor is primarily made for multithreaded workloads like web servers and JAVA application servers. The chip has a single floating-point unit for all 32 threads and hence its performance is poor for scientific applications.

The last studied system is a pure 32-way simulated CMP. This CMP has been configured using the Vasa simulator framework [16], which is based on the Simics full system simulator [12]. The simulator can run the same unmodified workloads as the previous systems. The simulated pure CMP is configured to resemble the UltraSPARC T1 processor except that it has no SMT-cores. On the simulated system, each thread has a separate pipeline. However, similar to the UltraSPARC T1 processor, a total of eight 8 KB first level caches are available in the system and each first level cache is shared between four threads. The second level cache is somewhat larger in the simulated system, 4 MB, than in the UltraSPARC T1, 3 MB, because of constraints in the simulator. The cache and memory latencies of the simulated machine was also chosen according to the latencies of the UltraSPARC T1. The simulated CMP has a separate floating-point unit for each thread.

8.1 Scalability on a cc-NUMA system

The first scalability experiment was to compare the performance of the RBGS- and TBGS-smoothers on the cc-NUMA system. Figure 7 shows the performance ratio per step between the TBGS- and RBGS-smoothers for the $N = 257$ problem size at different number of threads. The ratio is about 1.6 for the TBGS1-smoother compared to the RBGS-smoother for a single threaded run. This ratio means that if the TBGS-smoother would take one second per step, the RBGS-smoother would require 1.6 seconds to perform a step. When σ is increased, the performance ratio increases. The small number of second level cache misses makes the TBGS-smoothers with large values of σ very competitive especially since the memory access latency on this cc-NUMA system is large. However, the more threads that are used, the lower the ratio becomes. This is caused by an increased communication overhead in the TBGS-smoother. An interesting behavior in Figure 7 is that for $\sigma = 8$ and $\sigma = 16$ the performance ratio is lower for four threads or less. The active region does not entirely fit in the second level cache for these problems.

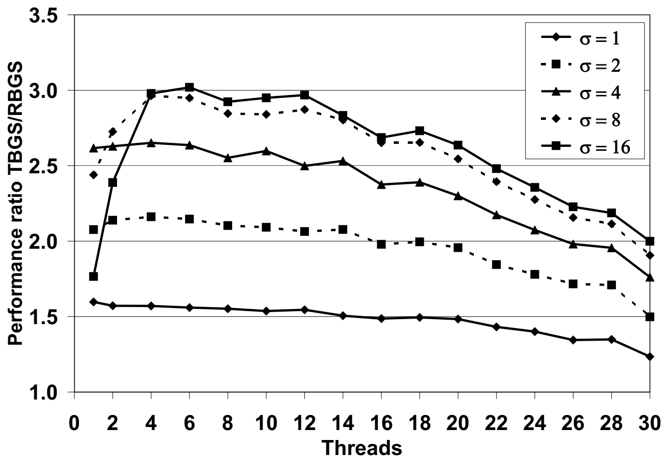


Figure 7. Performance ratio per step between the TBGS- and RBGS-smoothers at different number of threads for N=257. The experiments were performed on the 32-way cc-NUMA system.

The speedup for the RBGS-smoother relative the sequential RBGS-smoother is 39.4 on 30 threads for this problem size. The reason for the super-linear speedup is probably that the total size of the level two caches of all processors becomes larger when many threads are used. As mentioned in Section 3, RBGS could in the sequential case be optimized using blocking. Therefore the speedup would have been less if we compared the best cache optimized sequential RBGS-smoother with the original parallelized RBGS-smoother.

The TBGS16-smoother is about twice as fast as the RBGS-smoother at 30 threads. The conclusion is that the TBGS-smoother is a competitive choice also on traditional shared-memory machines since it scales well up to at least 30 threads. For even larger number of threads, the RBGS-smoother could be a better choice.

8.2 Scalability on CMPs

We continue with a study of the scalability of the TBGS-smoother on two types of CMPs. Our first study was performed on the UltraSPARC T1 processor. We edited the TBGS- and RBGS-smoothers to operate on 8 B integer values instead of 8 B floating-point values because of the processors lack of efficient floating-point computations. This makes the algorithms totally irrelevant for scientific purposes. However, we can still study the algorithms from a scalability point of view. As previously mentioned, the UltraSPARC T1 processor has eight 4-way SMT-cores. We have found that the SMT-ness of the processor has a significant influence on the performance for this problem. Therefore, we have made an additional study of a simulated 32-way pure CMP without SMT-cores. This study makes it possible to distinguish performance influences caused by the processor being a mixed SMT/CMP processor from the pure CMP case. All experiments have been carried out on floating-point values on the simulated machine.

In Figure 8, the performance ratio per step is compared for the RBGS- and the TBGS1-smoother at the $N = 129$ problem size. The figure shows both the performance ratio for the UltraSPARC T1 and the simulated pure CMP. For both processors, the threads are bound in a round-robin fashion to the cores. That is, the first eight threads are bound to the different cores and when additional threads are added, they are equally divided between the cores.

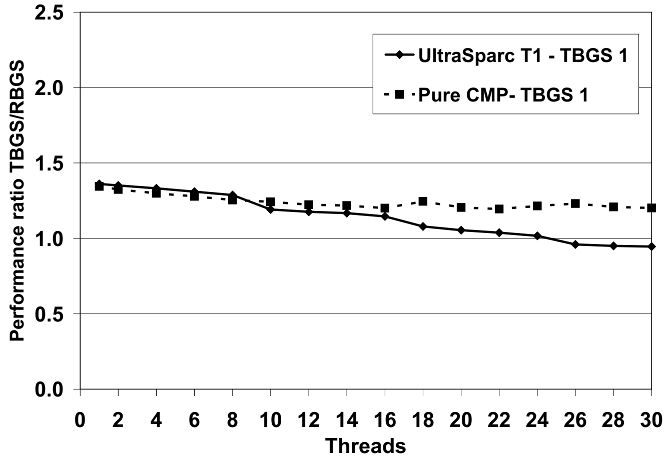


Figure 8. Performance ratio per step between TBGS and RBGS at different number of threads for N=129. The experiments were performed on the 32-way UltraSPARC T1 and the simulated 32-way pure CMP.

The performance ratio of the simulated CMP is larger than for the UltraSPARC T1 when many threads are used. The reason for this behavior is that the UltraSPARC T1 has SMT-cores. Since the figure shows the ratio between two smoothers that should perform basically the same type of work, this cannot be an effect of several threads contending the pipeline within a single core. The main difference between the TBGS1- and the RBGS-smoothers is as explained earlier that the RBGS-smoother causes about twice as many second level cache misses. The UltraSPARC T1 cores are designed for hiding memory latencies by performing other useful work during the memory accesses. The processor manages better to hide latencies for the RBGS- than the TBGS1-smoother since there are twice as many memory accesses. Therefore, the ratio becomes smaller when several threads are used within every core. In the pure CMP-case, a memory access will simply stall the pipeline until the data comes back. No other useful work is done during the stall. The stall time will be proportional to the number of cache misses for both smoothers.

The UltraSPARC T1 curve in Figure 8 also shows distinct steps at 8, 16 and 24 threads. This is an effect of uneven work distribution between the cores. The first step occurs when 10 threads are used. In this case, two cores will have two threads executing, while the rest of the cores only executes a single thread. Hence, the cores with two threads executing will dominate the total execution time. Once again, the SMT-cores influence the execution. In the RBGS-smoother, more second level cache misses occur than in the TBGS1-smoother, and the SMT-cores manage better to hide the RBGS memory latencies. At 8, 16 and 24 threads, the number of threads per core is well balanced and hence the TBGS-smoother performs best relative to the RBGS-smoother.

The SMT/CMP performance discussion does not explain how the RBGS- smoother could show better performance than the TBGS1-smoother at more than 24 threads (performance ratio < 1). The reason for this behavior is that the TBGS1-smoother has a start time overhead according to Section 4. This overhead becomes larger the more threads that are used. For this small problem size, $N = 129$, the effect is significant. The start time overhead is also the reason for the slope at each step of the UltraSPARC T1 curve in Figure 8. If we compensate for the start time overhead, we could get a measure of the communication cost of the smoothers. The ratio for the UltraSPARC T1 processor at 30 threads is 1.05 and for the pure CMP 1.34 if we compensate for the start time overhead. The ratio is for both processors 1.34 for a single thread. The conclusion is that the cost of communication is negligible for the TBGS1-smoother on a CMP. However, the scalability of the TBGS-smoother is negatively influenced by two other factors: (1) the start time overhead and (2) the SMT-design of the UltraSPARC T1 processor.

Figure 9 shows the performance ratio between the RBGS- and the TBGS-smoothers, when σ is varied on the UltraSPARC T1 processor. Also on this processor, a larger σ leads to fewer cache misses and therefore the performance of TBGS becomes better. However, since the memory access latency is much smaller on the UltraSPARC T1 server than on traditional systems such as a cc-NUMA, the effect of increasing σ is smaller. The difference between for example TBGS1 and TBGS2 should also decline for a large number of threads since the SMT-design manages to hide memory latencies better for applications with more memory accesses. This is difficult to conclude from Figure 9. The reason is once again the start time overhead. The larger the value of *sigma*, the smaller the start time overhead becomes. Therefore all the TBGS1, TBGS2 and TBGS4 ratio curves would meet at about 1.1 if we would have compensated for the start time overhead.

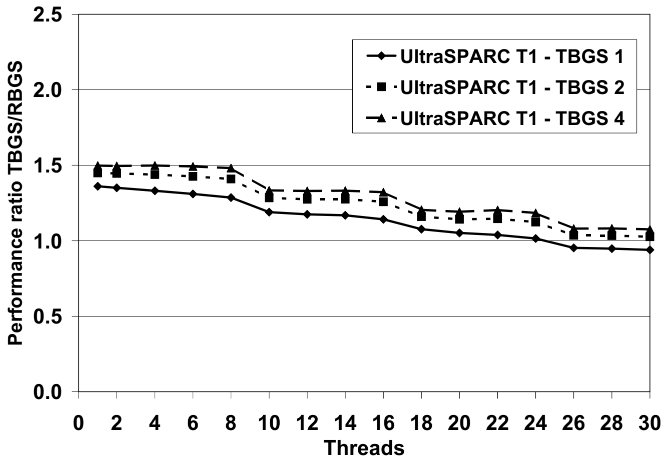


Figure 9. Performance ratio per step between TBGS and RBGS at different number of threads for N=129. The experiments were performed on the 32-way UltraSPARC T1.

We conclude the discussion on CMPs by comparing the speedup curves for the RBGS- and TBGS1-smoothers on the UltraSPARC T1 and the simulated pure CMP in Figure 10. The speedup curve is computed relative the performance of the sequential RBGS-smoother for each processor type. The speedup is best for the TBGS1-smoother on the pure CMP. The rather poor speedup of both the RBGS- and the TBGS1-smoother on the UltraSPARC T1 processor at more than 24 threads, is probably caused by a lack of resources in the pipeline when it is being used by four threads.

It should be noted that CMP-designs have greater tendencies to suffer from memory bandwidth limitations than single-core processors. The reason is that all threads in a CMP compete for the

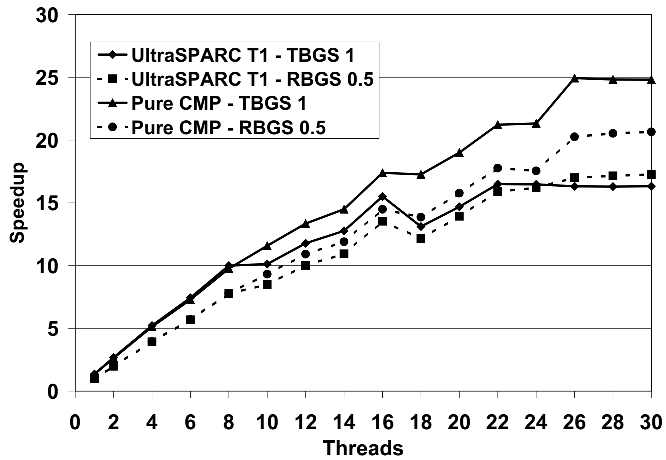


Figure 10. Speedup curves of the TBGS1- and RBGS-smoothers on the 32-way UltraSPARC T1 processor and the simulated 32-way pure CMP.

same bandwidth resources. Therefore it is more important to lower the number of memory references on a CMP. However, in our experiments we do not saturate the memory bandwidth of the UltraSPARC T1 processor. The reason is that the processor has an unusually high memory bandwidth of 25.6 GB/s. If the memory bandwidth had not been enough, the TBGS-smoother would have had a much larger performance advantage compared to the RBGS-smoother.

Many larger multiprocessor systems have earlier been made from coherent multiprocessor nodes, which have been connected using fast non-coherent interconnects. In the future, each node is probably built from chip multiprocessors. On these systems, a possible parallelization approach would be to use a cache optimized but communication-intensive smoother within each node and use a conventional parallelization technique between nodes. This could be implemented by mixing shared-memory programming on each node and message-passing programming between the nodes. An example of block-based parallelization methods have been presented by Block et al [4].

9 Multigrid Acceleration

As remarked already in Section 2, smoothing steps are in general not used in the simple form given by Algorithm 1. Instead, they are employed within a multigrid method. The basic idea of this type of scheme is to apply the smoother not only at the finest grid (level m), but also use it for computing corrections at a sequence of coarser grids defined by levels $l = 1, \ldots, m$. The smoother S_l is used to reduce a given component of the representation of the error, usually the high frequency part, at level l. For transferring values between grid levels, the *restriction* and *prolongation* operations R_l and P_l, have to be defined. As for the difference operators L_l, these are data parallel stencil operations.

The most fundamental multigrid scheme consists of replacing the single smoothing operation in Algorithm 1 with a (full-depth) *v-cycle*. In this case, the grid hierarchy is recursively descended until only a single, scalar equation remains. When this has been solved, the correction is step-by-step prolonged to finer grids and added to the current solution. This results in Algorithm 2. In the sequel, we

Algorithm 2 Multigrid_vcycle(l,u_l,f_l)

```
u_l, f_l, w_l, d_l, v_l :: grid functions at level l
S_l, L_l, P_l, R_l :: stencil operations at level l
if l = 1 then
    solve the scalar equation L_1 u_1 = f_1 and return
else
    w_l = S_l^γ(u_l, f_l) {Apply the smoother γ times}
    d_l = L_l w_l − f_l {Compute the defect}
    d_{l−1} = R_l d_l {Restrict the defect}
    v_{l−1} = 0 {Starting guess}
    call Multigrid_vcycle(l − 1, v_{l−1}, d_{l−1})
    v_l = P_l v_{l−1} {Prolong the coarse grid}
    u_l = u_l − v_l {Correct the solution}
    u_l = S_l^γ(u_l, f_l) {Apply the smoother γ times}
end if
```

use the term *iteration* for counting the number of v-cycles. Note that in Algorithm 2, the number of smoothing steps γ is normally a small constant. In fact, in many existing application codes $\gamma = 1$ is regularly used.

If a d-dimensional problem is solved, the grid at level l, $l = 1,\ldots,m$ contains N_l^d grid points where in the simplest case $N_l = 2^l - 1$. Since all operators involve only a set of neighboring points, it is clear that $O(N_l^d)$ arithmetic operations are required on each level. For each coarsening of the grid, the number of grid points is reduced by a factor 2^d, and a simple calculation shows that the arithmetic requirement for the full v-cycle is still $O(N_l^d)$. For important classes of PDEs, it can be proved that standard smoothers, restrictions and prolongations result in an multigrid method that converges in $O(1)$ iterations, independently of N_m, see e.g. Wesseling [18]. Thus, in these cases the multigrid scheme has optimal arithmetic requirements.

When used for a single grid, as described in Algorithm 1, it was already in [19] proved that the *asymptotic* convergence rate for the natural and red-black Gauss-Seidel schemes is equal for the model problem considered here. However, when applied as smoothers in the multigrid algorithm 2, then the red-black ordering results in slightly faster convergence [18]. In [18], it is also stated that for large problems, a multigrid scheme with a red-black Gauss-Seidel smoother results in possibly the most efficient solver for the Poisson equation in terms of the number of arithmetic operations performed.

Except for the smoother, all operations in Algorithm 2 are data parallel. Since the red-black Gauss-Seidel smoother has the smallest arithmetic requirements and also presents a high degree of parallelism, this has been the standard scheme in applications for some time. We have implemented multigrid methods using both our temporally blocked, parallelized naturally ordered smoother and the standard red-black scheme. The v-cycle iteration was put inside a parallel region, which implies that the data dependencies have to be protected by global barriers. The data parallel operations were parallelized using OpenMP work-sharing constructs. For the coarsest grids, the amount of data will be very small. This leads to increased parallel overhead compared to the arithmetical work [5], and eventually to reduced load balance. However, since the time spent computing on the coarsest grids is many magnitudes smaller than the time spent on the finest grids, the effect will usually be negligible.

10 Performance of a Multigrid Solver

The final study made in this paper is to study the performance of the multigrid solvers for the Poisson equation, which use the RBGS- and TBGS-smoothers. Once again, all experiments have been performed on the 8-way UltraSPARC IV+ CMP/SMP system. The execution time of the multigrid solver is dominated by the smoothing operation [15]. The multigrid solvers are run until convergence, which was defined to be that the norm of the solution should be $10^{-6}(\varepsilon)$ of the original norm.

In Section 9, it was remarked that the red-black smoother normally has somewhat better convergence rate than the temporally blocked smoother when used within a multigrid solver. A faster convergence rate could also be achieved by increasing the number of smoothing steps at each level, called γ in Algorithm 2. Table 3 shows the number of multigrid v-cycles that is required for convergence for the RBGS- and TBGS-smoothers using different numbers of steps γ.

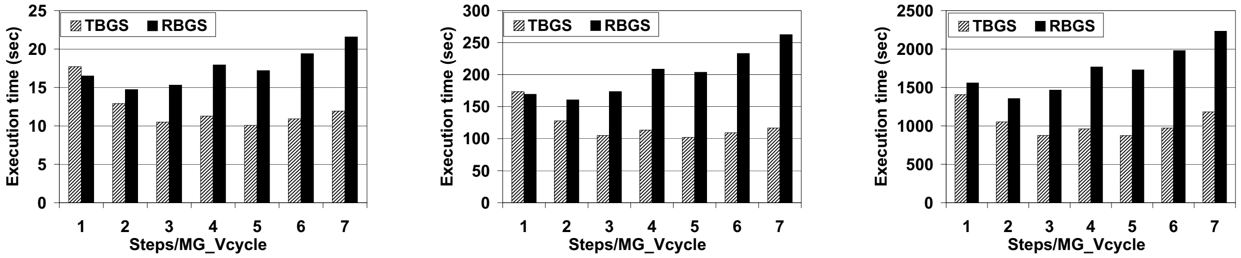
	γ	**1**	**2**	**3**	**4**	**5**	**6**	**7**
RBGS	**N=129**	11	8	7	7	6	6	6
	N=257	11	8	7	7	6	6	6
	N=513	12	8	7	7	6	6	6
TBGS	**N=129**	13	9	7	7	6	6	6
	N=257	13	9	7	7	6	6	6
	N=513	13	9	7	7	6	6	6

Table 3. Number of required multigrid v-cycles to reach convergence for different values of γ.

In Figure 11, the sequential execution time until convergence is presented for the multigrid solvers using the RBGS- and TBGS-smoothers for different values of γ. The results are presented for three different problem sizes, $N = 129$, $N = 257$ and $N = 513$. The results show that the solver with the TBGS-smoother and $\gamma = 3$ or $\gamma = 5$ performs best for all problem sizes. When the RBGS-smoother is employed, the best performance is obtained for $\gamma = 2$. The reason for the better performance of the TBGS multigrid solver is that the cost of performing additional smoothing vsteps is smaller in the TBGS- than the RBGS-smoother.

We parallelized the multigrid solver with two types of smoothers, RBGS and TBGS, using OpenMP. Note that no performance optimizations, such as blocking, have been made to other parts of the multigrid code than the smoothing steps. However, as mentioned earlier the execution time is dominated by this operation.

In Figure 12, the normalized execution time until convergence of the parallel solvers is presented. Three problem sizes are run with 1, 2, 4 and 8 processors. All execution times are normalized relative the sequential RBGS-smoother multigrid solver. The best values of γ is used both for RBGS ($\gamma = 2$) and TBGS ($\gamma = 5$). The best values stayed the same independent of the number of threads used for both solvers. The results show that the parallel TBGS multigrid solver is the fastest for the $N = 257$ and $N = 513$ problem size. In the $N = 129$ case, the parallel RBGS multigrid solver performs better. At this problem size, the communication cost is the largest and at more than two threads, the entire problem fits into the third level caches of the processors. Hence, the superior cache memory behavior of the TBGS-smoother compared to the RBGS-smoother is not exploited. The relative speedup of the fastest TBGS multigrid solver compared to the fastest RBGS multigrid solver is presented in Table 4. Translating the speedup into percentages, the TBGS

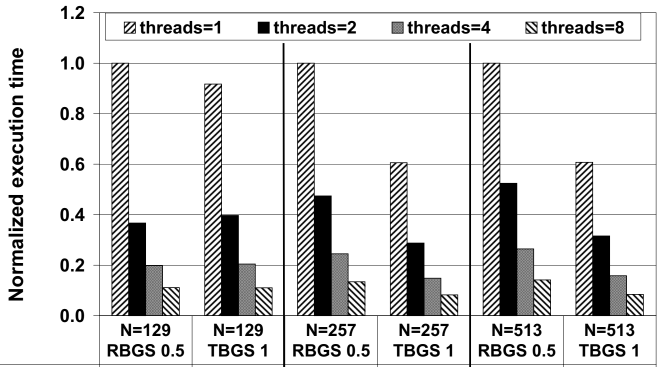


(a) Sequential execution time until convergence for $N = 129$ depending on the number of steps per multigrid v-cycle.

(b) Sequential execution time until convergence for $N = 257$ depending on the number of steps per multigrid v-cycle.

(c) Sequential execution time until convergence for N=513 depending on the number of steps per multigrid v-cycle.

Figure 11. Sequential performance of the multigrid solvers with RBGS- and TBGS-smoothers on the 8-way CMP/SMP UltraSPARC IV+ system.



	N=129 RBGS 0.5	N=129 TBGS 1	N=257 RBGS 0.5	N=257 TBGS 1	N=513 RBGS 0.5	N=513 TBGS 1
threads=1	1.000	0.917	1.000	0.606	1.000	0.607
threads=2	0.367	0.397	0.474	0.287	0.524	0.315
threads=4	0.198	0.205	0.245	0.148	0.264	0.158
threads=8	0.111	0.110	0.134	0.082	0.141	0.084

Figure 12. Normalized execution time of the parallel multigrid solvers with RBGS- and TBGS-smoothers for N=129, N=257 and N=513 on the 8-way CMP/SMP UltraSPARC IV+ system. The execution time is normalized relative the single processor RBGS multigrid solver.

threads	N=129	N=257	N=513
1	1.46	1.57	1.55
2	0.96	1.59	1.58
4	0.86	1.60	1.66
8	0.90	1.62	1.63

Table 4. Relative speedup of the multigrid solver with TBGS-smoothing compared to the RBGS multigrid solver on the 8-way CMP/SMP UltraSPARC IV+ system.

multigrid solver is about 40 percent faster than the RBGS multigrid solver for $N = 257$ and $N = 513$. For $N = 129$, the performance is somewhat worse for the TBGS multigrid solver than the RBGS multigrid solver.

11 Concluding Remarks

The paper presents a parallelization technique for the natural Gauss-Seidel ordering used as a smoother within a multigrid solver. The technique called parallel temporally blocked Gauss-Seidel exploits temporal cache locality much better than the normally used parallelization technique based on the red-black ordering. The temporally blocked smoother makes it possible to perform more than one step at a very low additional cost, since the temporal locality is preserved.

The temporally blocked Gauss-Seidel smoother has a larger communication cost than the red-black smoother, but it is compensated by its superior cache behavior. On a shared-memory multiprocessor with four two-way chip multiprocessors, the Poisson equation can be solved about 40 percent faster using a multigrid method based on the temporally blocked Gauss-Seidel smoother instead of the red-black smoother.

On future computer systems with chip multiprocessors, the communication cost will be decreased compared to recent systems, since synchronization can be made using a shared second level cache. We show that on a 32-way chip multiprocessor the communication cost is negligible and hence the scalability is good also for a communication-intense algorithm.

The low cost of synchronization within chip multiprocessors should encourage researchers to re-evaluate also other algorithms. For example, when standard ILU- and IC-factorizations are used for preconditioning in, e.g., conjugate gradient schemes, exactly the same type of stencil operations and data dependencies as for the Gauss-Seidel smoother appear. Also in these cases, the traditional parallel algorithms use multicolor orderings to avoid frequent synchronization. The results presented in this paper indicate that when the schemes are implemented on modern architectures, this may not longer be the best choice for maximum performance.

12 References

[1] L. M. Adams and J. M. Ortega. A Multi-Color SOR Method for Parallel Computation. In *Proceedings of the International Conference on Parallel Processing*, pages 53–58, 1982.

[2] L. A. Barroso, K. Gharachorloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing. In *Proceedings of the International Symposium on Computer Architecture*, pages 282–293, 2000.

[3] E. Berg and E. Hagersten. StatCache: A Probabilistic Approach to Efficient and Accurate Data Locality Analysis. In *Proceedings of the International Symposium on Performance Analysis of Systems and Software*, pages 20–27, 2004.

[4] U. Block, A. Frommer, and G. Mayer. Block Colouring

Schemes for the SOR Method on Local Memory Parallel Computers. *Parallel Computing*, 14:61–75, 1990.

[5] E. Chow, R. Falgout, J. Hu, R. Tuminaro, and U. Yang. A Survey of Parallelization Techniques for Multigrid Solvers. Technical Report UCRL-BOOK-205864, LLNL, 2004.

[6] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen. Simultaneous Multithreading: A Platform for Next-Generation Processors. *IEEE Micro*, 17(5):12–19, 1997.

[7] T. Kim and C.-O. Lee. A Parallel Gauss-Seidel Method using NR Data Flow Ordering. *Applied Mathematics and Computation*, 99(2):209–220, 1999.

[8] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: A 32-Way Multithreaded SPARC Processor. *IEEE Micro*, 25(2):21–29, 2005.

[9] K. Krewell. Power5 Tops on Bandwidth. *Microprocessor Report*, December 22, 2003.

[10] K. Krewell. Double Your Opterons; Double Your Fun. *Microprocessor Report*, October 11, 2004.

[11] K. Krewell. Sparc Turns 90 nm. *Microprocessor Report*, October 25, 2004.

[12] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Högberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A Full System Simulation Platform. *IEEE Computer*, 35(2):50–58, 2002.

[13] J. McGregor. Ringside for 2006 Dual-Core Fights. *Microprocessor Report*, December 19, 2005.

[14] N. R. Patel and H. F. Jordan. A Parallel Point Rowwise Successive Over-Relaxation Method on a Multiprocessor. *Parallel Computing*, 1:207–222, 1984.

[15] S. Sellappa and S. Chatterjee. Cache-Efficient Multigrid Algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[16] D. Wallin, H. Zeffer, M. Karlsson, and E. Hagersten. Vasa: A simulator infrastructure with adjustable fidelity. In *Proceedings of the 17th IASTED International Conference on Parallel and Distributed Computing and Systems*, 2005.

[17] C. Weiss, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proceedings of the Conference on Supercomputing*, page 31, 1999.

[18] P. Wesseling. *An Introduction to Multigrid Methods*. John Wiley and Sons Ltd., Chichester, 1992.

[19] D. M. Young. *Iterative Solution of Large Linear Systems*. Academic Press, New York, 1971.