Enhancing the Performance of Multigrid Smoothers in Simultaneous Multithreading Architectures *

Carlos García[1], Manuel Prieto[1], Javier Setoain[1], and Francisco Tirado[1]

Dto. Arquitectura de Computadores y Automática
Universidad Complutense de Madrid
Avd. Complutense s/n, 28040 Madrid, Spain
{garsanca,mpmatias,jsetoain,ptirado}@dacya.ucm.es

Abstract. We have addressed in this paper the implementation of red-black multigrid smoothers on high-end microprocessors. Most of the previous work about this topic has been focused on cache memory issues due to its tremendous impact on performance. In this paper, we have extended these studies taking *Simultaneous Multithreading* (*SMT*) into account. With the introduction of *SMT*, new possibilities arise, which makes highly advisable a revision of the different alternatives. A new strategy is proposed that focused on inter-thread sharing to tolerate the increasing penalties caused by memory accesses. Performance results on an *IBM's Power5* based system reveal that our alternative scheme can compete with and even improve sophisticated schemes based on tailored loop fusion and tiling transformations aimed at improving temporal locality.

1 Introduction

Multigrid methods are regarded as being the *fastest* iterative methods for the solution of the linear systems associated with elliptic partial differential equations, and as amongst the *fastest* methods for other types of integral and partial differential equations [16]. *Fastest* refers to the ability of Multigrid methods to attain the solution in a computational work which is a small multiple of the operation counts associated with discretizing the system. Such efficiency is known as *textbook multigrid efficiency* (TME) [15] and has made multigrid one of the most popular solvers on the niche of large-scale problems, where performance is critical.

Nowadays, however, the number of executed operations is only one of the factors that influences the actual performance of a given method. With the advent of parallel computers and superscalar microprocessors, other factors such as *inherent parallelism* or *data locality* (i.e. the memory access behavior of the algorithm) have also become relevant. In fact, recent evolution of hardware has exacerbated this trend since:

* This work has been supported by the Spanish research grants TIC 2002-750 and TIN 2005-5619

– The disparity between processor and memory speeds continues to grow despite the integration of large caches.
– Parallelism is becoming the key of performance even on high-end microprocessors, where multiple cores and multiple threads per core are becoming mainstream due to clock frequency and power limitations.

In the multigrid context, these trends have prompted the the development of specialized multigrid-like methods [1, 2, 10, 5], and the adoption of new schemes that try to bridge the processor/memory gap by improving locality [14, 18, 7, 4, 8]. Our focus in this paper is the extension of this cache-aware schemes to *Simultaneous Multithreading (SMT)* processors.

As its name suggests, *SMT* architectures allows several independent threads to issue instructions simultaneously in a single cycle [17]. Its main goal is to yield better use of the processor's resources, hiding the inefficiencies caused by long operational latencies such as memory accesses. At first glance, these processors can be seen as a set of logical processors that share some resources. CWith HT, the Intel Pentium 4 behaves as two logical processors sharing some resources (Functional Units, Memory Hierarchy, etc). The exploitation of this additional level of parallelism has been performed in this work by means of OpenMP directives, which are directly supported by the Intel ICC compileronsequently, one may think that optimizations targeted for *Symmetric Multiprocessors* (*SMP*) systems are also good candidates for *SMT*. However, unlike *SMP* systems, *SMT* provides and benefits from fine-grained sharing of processor and memory resources. On the other hand, unlike conventional superscalar architectures, *SMT* exposes and benefits from thread level parallelism when hiding latencies. Therefore, optimizations that are appropriate for these conventional machines may be inappropriate or less effective for *SMT* [9].

Unfortunately, *SMT* potentials are not yet fully exploited in most applications due to the relative underdevelopment of compilers, which despite many improvements still lag far behind. Due to this gap between compiler and processor technology, applications cannot benefit from *SMT* hardware unless they are explicitly aware of thread interactions. In this paper, we have revisited the implementation of multigrid smoothers in this light. The popularity of multigrid makes this study of great practical interest. In addition, it also provides certain insights about the potential benefits of this relatively new capability and how to take advantage of it, which could ideally help to develop more efficient compiler schemes.

The organization of this paper is as follows. We begin in Sections 2 and Section 3 by briefly introducing multigrid methods and describing the main characteristics of our target computing platform respectively. In Section 4 we describe the baseline codes used in our study for validation and assessment. They are based on the *DIME* project (*DIME* stands for *Data Local Iterative Methods For The Efficient Solution of Partial Differential Equations*) [3, 14, 18, 7, 4, 8], which is one of the most outstanding and systematic studies about the optimization of multigrid smoothers. Afterwards, in Section 5, we discuss our *SMT*-aware im-

plementation. Performance results are discussed in Section 6. Finally, the paper ends with some conclusions and hints for future research.

2 Multigrid Introduction

This section provides a brief introduction about multigrid, defining basic terms and describing the most relevant aspects of these methods so that we have a basis on which to discuss some of the performance issues.

The fundamental idea behind Multigrid methods [16] is to capture errors by utilizing multiple length scales (multiple grids). They consist of the following complementary components:

- *Relaxation.* The relaxation procedure, also called smoother in multigrid lingo, is basically a simple (and inexpensive) iterative method like *Gauß-Seidel*, damped *Jacobi* or block *Jacobi*. Its election depends on the target problem, but if well chosen, it is able to reduce the high-frequency or oscillatory components of the error in relatively few steps.
- *Coarse-Grid Correction.* Smoothers are ineffectual in attenuating low-frequency content of the error, but since the error after relaxation should lack the oscillatory components, it can be well-approximated using a coarser grid. On that grid, errors appear more oscillatory and thus the smoother can be applied effectively. New values are transferred afterwards to the target grid to update the solution.

The *Coarse-Grid Correction* can be applied recursively in different ways, constructing different cycling strategies. Algorithm 1 shows the pseudo-code of one of the most popular choices, known as V-cycle due to its pattern. This algorithm telescopes down to a given coarsest grid, and then works its way back to the target finest grid. The transfer operators I_h^{2h} and I_{2h}^h connect the grids levels: I_h^{2h} is known as the restriction operator and transfers values from a finer to a coarser level, whereas I_{2h}^h is known as the prolongation operator and maps from a coarser to a finer level.

Algorithm 1 V-cycle(ν_1,ν_2,v_h,b_h) multigrid V-cycle applied to the system $A_h u_h = b_h$ defined on a grid Ω_h.

if h == Coarsest **then**
  Return $u_H \leftarrow$ Solve(A_H,v_H,b_H)
else
  $v_h \leftarrow$ Smooth(ν_1,v_h,b_h)
  $b_{2h} \leftarrow I_h^{2h}(b_h - A_h v_h)$
  $v_{2h} \leftarrow$ V-cycle(ν_1,ν_2,0_{2h},b_{2h})
  $v_h \leftarrow v_h + I_{2h}^h(v_{2h}$
  Return $u_n \leftarrow$ Smooth(ν_2,v_h,b_h)
end if

The most time-consuming part of a multigrid method is the smoother and hence is the primary parameter in optimizing the solve time for a particular problem. In this initial study we have focused on point-wise smoothers. Block smoothers [12, 11] are more efficient in certain problems but are beyond the scope of this paper and will not be addressed at this time. The discussion about the implementation of point-wise smoothers is taken up in Section 4.

3 Experimental Platform

Our experimental platform consists in an *IBM's Power5* processor running under Linux, the main features of which are summarized in Table 1.

Table 1. Main features of the target computing platform.

Processor	IBM 2-way 1.5GHz Power5 (2 way core SMP)	
	L1 DataCache	32 KB 4-way associative, LRU
	L2 Unified Cache	1.9MB 10-way associative, LRU
	L3 Unified Cache (*off-chip*)	36MB shared per processor pair 10-way associative, LRU
Memory	2048 MBytes (4x512) DIMMS 266 MHz DDR SDRAM	
Operating System	GNU Debian Linux kernel 2.6.14-SMP for 64 bits	
IBM XL Fortran Switches (Advance Ed. v9.1)	-O5 -qarch=pwr5 -qtune=pwr5 -q64 -qhot -qcache=auto Parallelization with OpenMP: -qsmp=omp	

This processor has introduced *SMT* to the *IBM's Power* family [6]. With this design, each core of this dual-core processor appears to software as two logical CPUs, usually denoted as threads, that share some resources such as functional units or the memory hierarchy.

Apart from *SMT*, we should also highlight the impressive memory subsystem of the *Power5*. The memory controller is moved on chip and the main memory is directly connected to the processor via three buses: the address/command bus, the unidirectional write data bus, and the unidirectional read data bus. The 36-MB off-chip L3 has been removed from the path between the processor to the memory controller and operates as a victim cache for the L2. This means that data is transferred to the L3 only when it is replaced from the L2.

Finally, it is worth to mention that the exploitation of *SMT* has been performed in this work by means of *OpenMP* directives, which are directly supported by the IBM's FORTRAN compiler. Single thread performance have been

measured using a sequential code and enabling the *Power5 single-threaded* mode of the *Power5*, which gives all of the chip resources to one of the logical CPUs.

4 Cache-aware Red-Black Smoothers

Gauß-Seidel has long been the smoother of choice within multigrid on both structured and unstructured grids [1]. Although, it is inherently sequential in its natural form (the lexicographic ordering), it is possible to expose parallelism by applying multi-coloring, i.e. splitting grid nodes into disjoint sets, with each set having a different color, and updating simultaneously all nodes of the same color.

The best known example of this approach is the red-black Gauß-Seidel for the 5-point Laplace stencil, which is schematically illustrated in Figure 1. For the 9-point Laplacian, a red-black ordering may lead to a race condition (depending on the implementation), and at least a four color ordering of the grid space is needed to decouple the grid nodes completely.

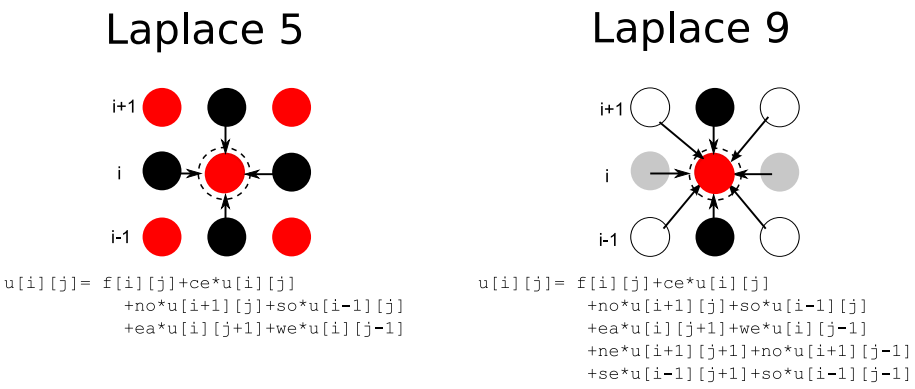

Fig. 1. 2D red-black Gauß-Seidel for the 5-point (on left-hand side) and 9-point (on right-hand side) Laplace stencils

Apart from exposing parallelism, multi-coloring also impacts on the convergence rate, but unlike other techniques such as block Gauß-Seidel (i.e. applying Gauß-Seidel locally on every processor), the overall multigrid convergence rates remain satisfactory.

Unfortunately, multi-coloring deteriorate the memory access and may lead to poor performance. Algorithm 4 shows the pseudo-code of a red-black smoother, denoted as *rb1* by the *DIME* project. This naïve implementation performs a complete sweep through the grid for updating all of the red nodes, and then another complete sweep for updating all of the black nodes. Therefore, *rb1* exhibits lower spatial locality than a lexicographic ordering. Furthermore, if the target grid is large enough, temporal locality is not fully exploited.

Algorithm 2 Red-Black Gauß-Seidel naïve implementation

```
for it=1,nIter do
  // red nodes:
  for i = 1; n-1 do
    for j = 1+(i+1)%2; n-1; j=j+2 do
      Relax_point( i,j )
    end for
  end for
  // black nodes:
  for i = 1, n-1 do
    for j = 1+i%2; n-1; j=j+2 do
      Relax_point( i,j )
    end for
  end for
end for
```

Alternatively, some authors have successfully improved cache reuse (locality) using loop reordering and data layout transformations that were able to improve both temporal and spatial data locality [13, 18].

Following these previous studies, in this paper we have used as baseline codes the different red-black smoothers developed within the framework of the *DIME* project. To simplify matters, these codes are restricted to 5-point as well as 9-point discretization of the Laplacian operator. Figures 2-4 illustrate some of them, which are based on the following observations:

- The black nodes of a given row $i-1$ can be updated once the red nodes of the i row has been updated. This is the idea behind the *DIME's rb2* (see figure 2) and *rb3* schemes, which improve both temporal and spatial locality fusing the *red* and *black* sweeps.
- If several successive relaxation have to be performed, additional improvements can be achieved transforming the iteration traversal so that the operations are performed on small 1D or 2D tiles of the whole array. Data within a tile is used as many times as possible before moving to the next tile. *DIME's rb4-rb9* schemes perform different 1D and 2D tiling transformations. Figures 3 and 4 illustrate the *rb5* and *rb9* schemes respectively.

Tables 2 and 3 show the MFlops achieved by *DIME's rb1-9* codes on our target platform. The speedup of the best transformation range from 1.2 to 1.8 for the 5-point stencil, and from 1.15 to 1.25 for the 9-point version. Our first insight is that these gains are lower than on other architectures. For instance, the improvements on a DEC PWS 500au reported on *DIME's* website reach a factor of 4 [3]. Furthermore, the sophisticated two-dimensional blocking transformation *DIME's rb9* does not provide additional improvements, being *DIME's rb7* and sometimes *DIME's rb2* the most effective transformations.

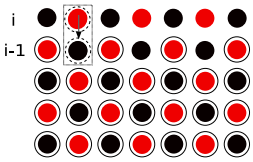

Fig. 2. *DIME's* rb2. The update of red and black nodes is fused to improve temporal locality.

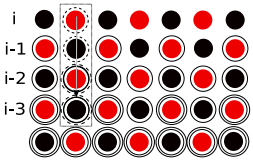

Fig. 3. *DIME's* rb5. Data within a tile is reused as much as possible before moving to the next tile.

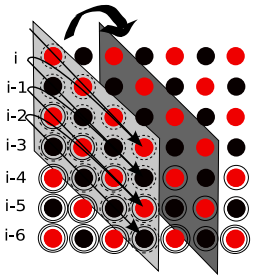

Fig. 4. *DIME's* rb9. Data within a tile is reused as much as possible before moving to the next tile.

The main reason behind this difference in behavior is the relatively large amount of on-chip and off-chip caches included in the *IBM's Power5*, as well as their higher degree of associativity.

5 SMT-aware Red-Black Smoothers

The availability of *SMT* introduces a new scenario in which thread-level parallelism can also be applied to hide memory accesses. As mentioned above, *SMT* processors can be seen as a set of logical processors that share execution units, systems buses and the memory hierarchy. This logical view suggests the application of the general principles of data partitioning to get the multithreaded versions of the different *DIME* variants of the red-black Gauß-Seidel smoother. This strategy, which can be easily expressed with *OpenMP* directives, is suitable for shared memory multiprocessor. However, in a *SMT* microprocessor, the similarities amongst the different threads (they execute the same code with just a different input dataset) may cause contention since they have to compete for the same resources.

Alternatively, we have employed a dynamic partitioning where computations are broken down into different tasks with are assigned to the pool of available threads. Intuitively, the smoothing of the different colors is interleaved by assigning the relaxation of each color to a different thread. This interleaving is controlled by a scheduler, which avoids race conditions and guarantees a deterministic ordering.

Algorithm 3 shows a pseudo-code of this approach for red-black smoothing. Our actual implementation is based on the *OpenMP's parallel* and *critical* directives. The critical sections introduce some overhead but are necessary to avoid race-conditions. However, the interleaving prompted by the scheduling allows the *black thread* to take advantage of some sort of data prefetching since it pro-

cesses grid nodes that have just been processed by the *red thread*, i.e. the *red thread* acts as a helper thread that performs data prefetching for the *black* one.

Algorithm 3 Interleaved implementation of a red-black Gauß-Seidel

```
#pragma omp parallel private(task,more_tasks) shared(control_variables)
more_tasks == true
while more_tasks do

  #pragma omp critical
  Scheduler.next_task(&task)

  if (task.type == RED) then
    Relax_RED_line(task);
  end if

  if (task.type == BLACK) then
    Relax_BLACK_line(task);
  end if

  #pragma omp critical
  more_task=Scheduler.commit(task)

end while
```

This interleaved approach can also be combined with *DIME's rb4-8* variants. If two successive iterations have to be performed, the intuitive idea is that one thread performs the first relaxation step whereas the other performs the second one. The scheduler guarantees again a deterministic ordering.

In the next section we compare the performance of this novel approach over traditional block-outer, cyclic-outer and cyclic-inner distributions of the relaxation nested loop. For the naïve implementation, all of them are straightforward. However, for *DIME's rb2-8* variants, the cyclic-outer version is non-deterministic, whereas the block-outer requires the processing of block boundaries in advance.

We have omitted a parallel version of *DIME's rb9* since even in the sequential setting, that version does not provide superior performance over *DIME's rb5-8*. We have also omitted block-outer and cyclic-outer distributions of the red-black smoother for the 9-point stencil, since they are also non-deterministic. Note, however, that both the cyclic-inner and our interleaved approach avoid race-conditions.

6 Performance Results

Figure 5 shows the speedup achieved by the different parallel strategies over the baseline code (with the best DIME's transformation) for the the 5-point stencil.

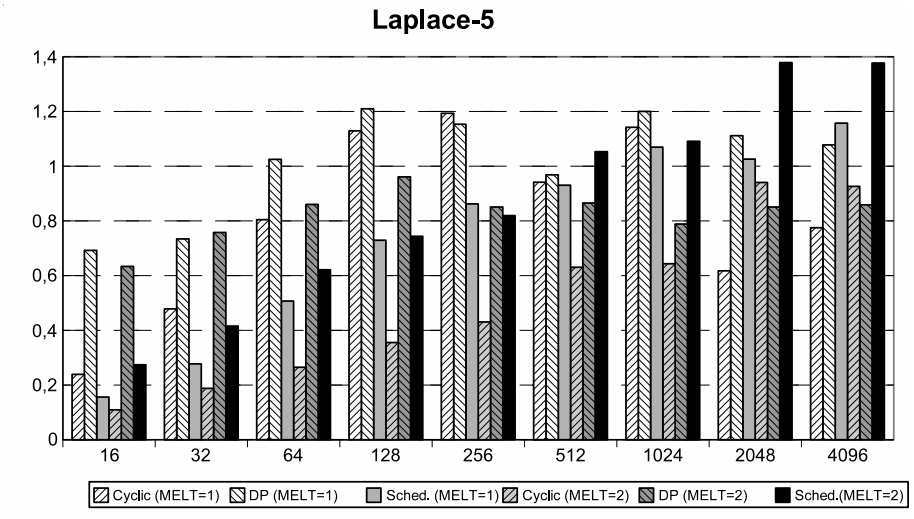

Fig. 5. Speedup achieved by different parallel implementations of a red-black Gauß-Seidel smoother for a 5-point Laplace stencil. Sched denotes our strategy, whereas DP and Cyclic denote the best block and a cyclic distribution of the smoother's outer loop respectively. MELT is the number of successive relaxations that have been applied.

As can be noticed, the election of the most suitable strategy depends on the grid size:

- For small and medium grid sizes block and cyclic distributions outperforms our approach, although for the smallest sizes none of them is able to improve performance. This is the expected behavior given that for small and medium working sets, memory bandwidth and data cache exploitation are not a key issue and traditional strategies beats our approach on performance due to the overheads introduced by the dynamic task scheduling.
- For large sizes we observe the opposite behavior given that the overheads involved in task scheduling become negligible, whereas the competition for memory resources becomes a bottleneck in the other versions. In fact, we should highlight that the block and cyclic distributions become clearly inefficient for large grids.

– The break-even point between the static distributions and our interleaved approach is a relative large grid due to the impressive L3 cache (36 MB) of the *Power5*.

Figure 6 confirms some of these observations for the the 9-point stencil. Furthermore, the improvements over *DIME's* variants are higher in this case, since this is a more demanding problem.

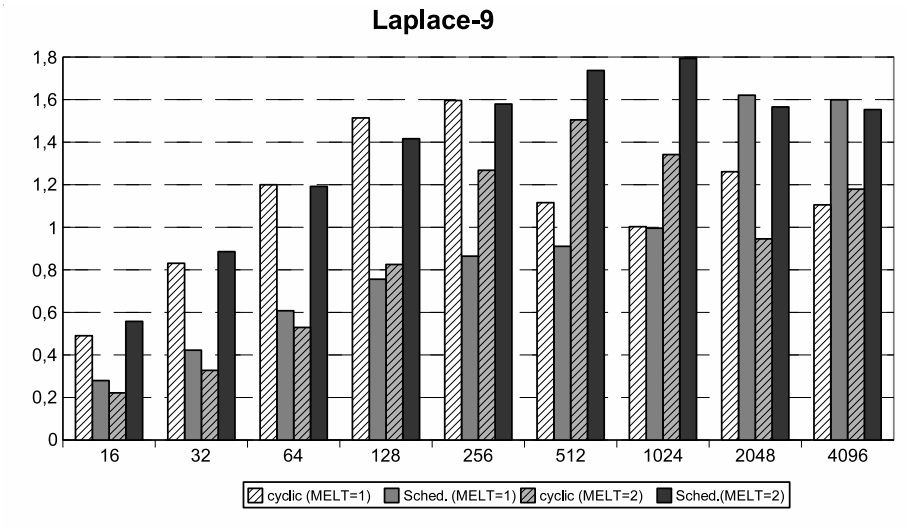

Fig. 6. Speedup achieved by different parallel implementations of a red-black Gauß-Seidel smoother for a 9-point Laplace stencil. Sched denotes our strategy, whereas Cyclic denotes the best cyclic distribution of the smoother's inner loop. MELT is the number of successive relaxations that have been applied.

7 Conclusions

In this paper, we have introduced a new implementation of red-black Gauß-Seidel Smoothers, which on *SMT* processors fits better than other traditional strategies. From the results presented above, we can draw the following conclusions:

– Our alternative strategy, which implicitly introduce some sort of tiling amongst threads, provide noticeable speed-ups that match or even outperform the results obtained with the different *DIME's rb2-9* variants for large grid sizes. Notice that instead of improving *intra-thread locality*, our strategy improves locality taking advantage of fine-grain thread sharing.

– For large grid sizes, competition amongst threads for memory bandwidth and data cache works against traditional block distributions. Our interleaved approach performs better in this case, but suffers important penalties for small grids, since its scheduling overheads does not compensate its better exploitation of the temporal locality. Given that multigrid solvers process multiple scales, we advocate hybrid approaches.

We are encouraged by these results, and based on what we have learned in this initial study we are proceeding with:

– Analyzing more elaborated multigrid solvers.
– Combining interleaving with grid partitioning distributions to scale beyond two threads. The idea is to use grid partitioning to distribute data amongst a large scale system, and interleaving to exploit thread level parallelism inside their cores.

References

1. M. F. Adams, M. Brezina, J. J. Hu, and R. S. Tuminaro. Parallel multigrid smoothing: polynomial versus Gauss-Seidel. *J. Comp. Phys.*, 188(2):593–610, 2003.
2. Edmond Chow, Robert D. Falgout, Jonathan J. Hu, Raymond S. Tuminaro, and Ulrike Meier Yang. A survey of parallelization techniques for multigrid solvers,. Technical report, 2004.
3. Friedrich-Alexander University Erlangen-Nuremberg. Department of Computer Science 10. DIME project. Available at http://www10.informatik.uni-erlangen.de/Research/Projects/DiME-new.
4. C.C. Douglas, J. Hu, M. Kowarschik, U. Rüde, and C. Weiß. Cache Optimization for Structured and Unstructured Grid Multigrid. *Electronic Transactions on Numerical Analysis (ETNA)*, 10:21–40, 2000.
5. F. Hülsemann, M. Kowarschik, M. Mohr, and U. Rde. Parallel geometric multigrid. *Lecture Notes in Computer Science and Engineering*, 51:165–208, 2005.
6. Ronald N. Kalla, Balaram Sinharoy, and Joel M. Tendler. IBM Power5 chip: A dual-core multithreaded processor. *IEEE Micro*, 24(2):40–47, 2004.
7. M. Kowarschik, U. Rüde, C. Weiß, and W. Karl. Cache-Aware Multigrid Methods for Solving Poisson's Equation in Two Dimensions. *Computing*, 64:381–399, 2000.
8. M. Kowarschik, C. Weiß, and U. Rüde. Data Layout Optimizations for Variable Coefficient Multigrid. In P. Sloot, C. Tan, J. Dongarra, and A. Hoekstra, editors, *Proc. of the 2002 Int. Conf. on Computational Science (ICCS 2002), Part III*, volume 2331 of *Lecture Notes in Computer Science (LNCS)*, pages 642–651, Amsterdam, The Netherlands, 2002. Springer.
9. Jack L. Lo, Susan J. Eggers, Henry M. Levy, Sujay S. Parekh, and Dean M. Tullsen. Tuning compiler optimizations for simultaneous multithreading. In *International Symposium on Microarchitecture*, pages 114–124, 1997.
10. W. Mitchell. Parallel adaptive multilevel methods with full domain partitions. *App. Num. Anal. and Comp. Math*, 1:36–48, 2004.
11. Manuel Prieto, Rubén S. Montero, Ignacio Martín Llorente, and Francisco Tirado. A parallel multigrid solver for viscous flows on anisotropic structured grids. *Parallel Computing*, 29(7):907–923, 2003.

12. Manuel Prieto, R. Santiago, David Espadas, Ignacio Martín Llorente, and Francisco Tirado. Parallel multigrid for anisotropic elliptic equations. *J. Parallel Distrib. Comput.*, 61(1):96–114, 2001.
13. D. Quinlan, F. Bassetti, and D. Keyes. Temporal locality optimizations for stencil operations within parallel object-oriented scientific frameworks on cache-based architectures. In *Proceedings of the PDCS'98 Conference*, July 1998.
14. U. Rüde. Iterative Algorithms on High Performance Architectures. In *Proc. of the EuroPar-97 Conf.*, Lecture Notes in Computer Science (LNCS), pages 26–29. Springer, 1997.
15. James L. Thomas, Boris Diskin, and Achi Brandt. Textbook multigrid efficiency for fluid simulations. *Annual Review of Fluid Mechanics*, 35:317–340, 2003.
16. U. Trottenberg, C. Oosterlee, and A. Schller. *Multigrid*. Academic Press, 2000.
17. Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous multithreading: Maximizing on-chip parallelism. In *25 Years ISCA: Retrospectives and Reprints*, pages 533–544, 1998.
18. C. Weiß, W. Karl, M. Kowarschik, and U. Rüde. Memory Characteristics of Iterative Methods. In *Proc. of the ACM/IEEE Supercomputing Conf. (SC99)*, Portland, Oregon, USA, 1999.

Table 2. MFlops achieved by the different *DIME's* variants of the red-black Gauß-Seidel for a 5-point Laplace stencil. MELT denotes the number of successive iterations of this smoother.

	16		**32**		**64**	
	no PAD.	PADDING	no PAD.	PADDING	no PAD.	PADDING
rb1 (MELT=1)	1220.41	1328.02	1554.55	1886.03	1846.53	2151.69
rb2 (MELT=1)	1654.44	1799.24	1590.53	2353.55	2396.09	2464.36
rb3 (MELT=1)	1655.28	1800.82	1588.90	2364.93	2387.07	2474.08
rb4 (MELT=2)	1317.74	1353.41	1418.75	1435.21	1468.09	1471.15
rb4 (MELT=3)	1375.39	1410.50	1447.29	1489.77	1368.22	1384.25
rb5 (MELT=2)	1315.34	1355.12	1418.45	1433.10	1462.85	1472.53
rb5 (MELT=3)	1371.65	1411.75	1446.33	1490.16	1364.34	1386.79
rb6 (MELT=2)	1834.07	2055.36	2139.22	2743.28	2961.18	3101.63
rb6 (MELT=3)	1851.40	2121.46	2137.32	2908.11	3138.46	3235.82
rb7 (MELT=2)	2079.85	2091.77	2406.82	2826.85	3052.28	3128.04
rb7 (MELT=3)	1728.44	1768.35	1916.17	2066.12	2030.17	2124.82
rb8 (MELT=2)	2061.58	2087.47	2511.27	2833.20	3052.79	3128.73
rb8 (MELT=3)	1726.98	1766.26	1871.81	2038.05	1998.76	2069.93
rb9 (MELT=4)	1603.72	1672.13	2075.09	2349.55	2453.04	2643.53

	128		**256**		**512**	
	no PAD.	PADDING	no PAD.	PADDING	no PAD.	PADDING
rb1 (MELT=1)	2637.98	2673.35	2801.20	2823.32	2139.93	2412.95
rb2 (MELT=1)	2642.36	2667.40	2794.10	2794.10	2178.04	2495.04
rb3 (MELT=1)	2661.24	2664.53	2840.70	2840.70	2150.20	2500.19
rb4 (MELT=2)	1368.15	1380.44	1370.40	1375.65	1248.74	1292.01
rb4 (MELT=3)	1386.74	1393.14	1359.37	1368.66	1377.90	1427.34
rb5 (MELT=2)	1367.39	1379.36	1371.12	1373.43	1248.87	1292.34
rb5 (MELT=3)	1388.05	1398.70	1359.08	1369.72	1386.58	1425.09
rb6 (MELT=2)	3362.78	3445.60	3453.84	3513.04	2009.28	2561.72
rb6 (MELT=3)	3677.55	3710.23	3376.07	3487.90	1372.93	2390.08
rb7 (MELT=2)	3461.82	3553.01	3597.04	3625.99	1976.21	2619.95
rb7 (MELT=3)	2190.06	2207.63	2038.09	2109.38	1332.34	1833.58
rb8 (MELT=2)	3488.76	3549.89	3553.04	3618.60	1995.86	2620.77
rb8 (MELT=3)	2124.59	2152.84	2033.73	2054.91	1355.92	1788.43
rb9 (MELT=4)	2785.56	2965.07	2262.49	2981.07	864.31	2079.22

	1024		**2048**		**4096**	
	no PAD.	PADDING	no PAD.	PADDING	no PAD.	PADDING
rb1 (MELT=1)	1530.51	1904.80	1012.56	1130.23	827.73	908.38
rb2 (MELT=1)	1135.85	2012.24	1108.30	1322.43	909.35	1060.96
rb3 (MELT=1)	1138.72	1995.31	1108.83	1311.42	909.32	1060.65
rb4 (MELT=2)	691.73	1366.76	645.31	1198.68	607.49	1048.54
rb4 (MELT=3)	699.54	1405.66	664.97	1283.75	625.90	1096.99
rb5 (MELT=2)	691.23	1364.57	644.51	1191.35	607.49	1047.26
rb5 (MELT=3)	700.39	1406.21	664.47	1283.65	625.21	1093.09
rb6 (MELT=2)	797.33	2405.67	424.95	1228.52	425.91	1202.53
rb6 (MELT=3)	740.90	2385.51	377.16	1046.89	359.09	985.14
rb7 (MELT=2)	769.53	2319.02	490.97	1336.46	465.97	1248.94
rb7 (MELT=3)	706.90	1658.36	598.65	1132.00	555.64	988.34
rb8 (MELT=2)	888.24	2507.34	460.97	1087.26	474.43	1193.49
rb8 (MELT=3)	713.10	1670.59	551.20	1201.43	464.93	954.55
rb9 (MELT=4)	444.54	1928.21	353.77	1248.75	348.30	1058.10

Table 3. MFlops achieved by the different *DIME's* variants of the red-black Gauß-Seidel for a 9-point Laplace stencil. MELT denotes the number of successive iterations of this smoother.

	16		**32**		**64**	
	no PAD.	PADDING	no PAD.	PADDING	no PAD.	PADDING
rb1 (MELT=1)	846.24	845.91	1134.27	1172.41	1321.73	1348.14
rb2 (MELT=1)	950.62	1034.2	1042.20	1414.47	1126.24	1656.04
rb3 (MELT=1)	962.94	964.20	1248.63	1278.89	1371.60	1464.29
rb4 (MELT=2)	746.65	747.42	760.19	764.49	740.99	748.75
rb4 (MELT=3)	780.77	783.89	782.92	793.98	752.32	770.89
rb5 (MELT=2)	623.14	627.04	602.06	607.46	561.58	564.05
rb5 (MELT=3)	662.97	670.53	624.34	636.21	586.11	598.19
rb6 (MELT=2)	796.40	802.01	837.04	848.68	835.16	848.87
rb6 (MELT=3)	777.82	977.49	1094.88	1159.87	1125.33	1173.56
rb7 (MELT=2)	791.24	819.63	809.41	852.67	813.76	821.87
rb7 (MELT=3)	916.94	932.90	1007.62	1038.20	1029.50	1066.33
rb8 (MELT=2)	818.48	846.73	857.17	883.96	858.65	868.43
rb8 (MELT=3)	919.03	936.28	1007.72	1038.38	1047.92	1084.02
rb9 (MELT=4)	817.50	837.66	937.08	983.44	1039.88	1101.31

	128		**256**		**512**	
	no PAD.	PADDING	no PAD.	PADDING	no PAD.	PADDING
rb1 (MELT=1)	1667.04	1670.97	1729.36	1759.88	1562.05	1554.85
rb2 (MELT=1)	1831.58	1925.38	2018.26	2067.22	1951.36	1941.23
rb3 (MELT=1)	1727.06	1747.52	1862.76	1857.99	1742.83	1715.75
rb4 (MELT=2)	698.13	699.38	696.54	696.83	719.67	731.25
rb4 (MELT=3)	765.17	766.85	755.06	753.95	727.05	737.16
rb5 (MELT=2)	564.48	564.48	558.92	560.02	545.81	548.25
rb5 (MELT=3)	585.55	585.71	575.29	576.74	560.88	564.15
rb6 (MELT=2)	864.13	863.54	868.21	868.07	675.26	836.98
rb6 (MELT=3)	1284.45	1286.39	1280.24	1293.26	765.69	1135.30
rb7 (MELT=2)	831.41	831.08	821.43	830.45	725.22	794.60
rb7 (MELT=3)	1083.47	1083.95	1076.12	1080.96	903.37	913.06
rb8 (MELT=2)	885.39	878.57	885.83	892.54	776.17	859.16
rb8 (MELT=3)	1104.18	1105.80	1099.24	1100.55	961.44	976.59
rb9 (MELT=4)	1130.08	1146.71	871.69	1147.63	532.22	868.32

	1024		**2048**		**4096**	
	no PAD.	PADDING	no PAD.	PADDING	no PAD.	PADDING
rb1 (MELT=1)	1209.56	1249.72	497.73	554.38	504.74	571.47
rb2 (MELT=1)	932.55	1656.93	374.89	521.46	420.05	574.23
rb3 (MELT=1)	1295.14	1468.16	583.47	671.95	662.89	632.52
rb4 (MELT=2)	679.67	717.89	593.96	659.83	593.07	663.24
rb4 (MELT=3)	691.21	726.90	611.44	681.95	609.99	684.88
rb5 (MELT=2)	386.53	554.37	366.12	535.87	367.82	533.03
rb5 (MELT=3)	388.97	561.67	373.21	542.73	374.81	546.64
rb6 (MELT=2)	361.31	594.64	274.79	566.75	273.87	567.97
rb6 (MELT=3)	317.80	552.96	285.76	449.79	274.89	445.53
rb7 (MELT=2)	436.52	653.60	340.72	685.85	359.81	570.45
rb7 (MELT=3)	618.65	1053.75	762.01	754.74	471.25	701.31
rb8 (MELT=2)	438.12	697.92	333.05	617.26	384.65	613.42
rb8 (MELT=3)	641.91	1073.55	760.33	795.94	497.73	663.63
rb9 (MELT=4)	339.64	896.80	290.39	617.66	278.66	573.70