Improving Geographical Locality of Data for Shared Memory Implementations of PDE Solvers

Henrik Löf, Markus Nordén and Sverker Holmgren
Uppsala University, Department of Information Technology
P.O. Box 337, SE-751 05 Uppsala, Sweden
{Henrik.Lof,Markus.Norden,Sverker.Holmgren}@it.uu.se

Abstract

On cc-NUMA multi-processors, the non-uniformity of main memory latencies motivates the need for co-location of threads and data. We call this special form of data locality, *geographical locality*, as the non-uniformity is a consequence of the physical distance between the cc-NUMA nodes. In this article, we compare the well established method of exploiting the first-touch strategy using parallel initialization of data to an application-initiated page migration strategy as means of increasing the geographical locality for a set of important scientific applications.

Four PDE solvers parallelized using OpenMP are studied; two standard NAS NPB3.0-OMP benchmarks and two kernels from industrial applications. The solvers employ both structured and unstructured computational grids. The main conclusions of the study are: (1) that geographical locality is important for the performance of the applications, (2) that application-initiated migration outperforms the first-touch scheme in almost all cases, and in some cases even results in performance which is close to what is obtained if all threads and data are allocated on a single node. We also suggest that such an application-initiated migration could be made fully transparent by letting the OpenMP compiler invoke it automatically.

1 Introduction

In modern computer systems, *temporal* and *spatial locality* of data accesses is exploited by introducing a memory hierarchy with several levels of cache memories. For large multiprocessor servers, an additional form of locality also has to be taken into account. Such systems are often built as cache-coherent, non-uniform memory access (cc-NUMA) architectures [8], where the main memory is physically, or *geographically* distributed over several multi-processor nodes. The access time for local memory is smaller than the time required to access remote memory, and the *geographical locality* of the data influences the performance of applications.

The NUMA-ratio is defined as the ratio of the latencies for remote to local memory. Currently, the NUMA-ratio for the commonly used large cc-NUMA servers ranges from 2 to 6 [12, 18, 6, 19, 24]. If the NUMA-ratio is large, improving the geographical locality may lead to large performance improvements. This has been recognized by many researchers, and the study of geographical placement of data in cc-NUMA systems has been an active research area for some years [2, 27, 4, 23, 1, 22, 3]. Also, some techniques aimed at improving geographical locality are commonly employed today, both in the design of computer systems and in the design of algorithms and codes.

Good geographical locality is often a key to scalable performance on cc-NUMA systems when a shared-memory parallelization model, such as OpenMP [9], is used [21, 20]. Future microprocessors will need to explore thread-level parallelism for improving performance, which means that shared-memory programming models and multi-threading will probably be widely accepted and used. For large-scale problems, the inherent parallelism in efficient algorithms often permits an increasing number of threads. However, when the number of threads, and/or the data set, becomes so large that a non-uniform architecture is required, the performance effects of geographical locality needs to be considered.

In this paper we examine how different data placement schemes affect the performance of two important classes of parallel codes from large-scale scientific computing. The main issues considered are:

- What impact does geographical locality have on the performance for the type of algorithms studied?
- How does the performance of an application-initiated data migration strategy based on a migrate-on-next-touch feature compare to that of standard data placement schemes?
- How should such an application-initiated migration strategy be invoked?

Most experiments presented in this paper are performed using a Sun Fire 15000 (SF15k) [6] system, which is a commercial cc-NUMA computer. Some experiments are also performed using a Sun WildFire prototype system [13, 14].

2 Geographical locality

Algorithms with static data access patterns can achieve good geographical locality by carefully allocating the data at the nodes where it is accessed. The standard technique for creating geographical locality is based on static first-touch page allocation implemented in the operating system. In a first-touch scheme, a memory page is placed at the node where its first page fault is generated. However, the first-touch scheme also has some well known problems. In most cases, the introduction of pre-iteration loops in the application code is necessary to avoid serial initialization of the data structures, which would lead to data allocation on a single node. For complex application codes, the programming effort required to introduce these loops may be significant. For other important algorithm classes, the access pattern for the main data structures is computed in the program. In such situations it may be difficult, or even impossible, to introduce pre-iteration loops in an efficient way. Instead, some kind of dynamic page placement strategy is required, where misplacement of pages is corrected during the execution by migrating and/or replicating pages to the nodes that perform remote accesses. Dynamic strategies might be explicitly initiated by the application[1], implicitly invoked by software [21], or they may be implicitly invoked by the computer system [27, 5, 7].

3 The PDE solvers

To evaluate different methods for improving geographical locality we study the performance of four solvers for large-scale partial differential equation (PDE) problems. In computational science, many important phenomena are modeled by PDEs, examples range from chemical reactions and the flow around an airplane to the prices of stock options. In the discretization of a PDE, a grid of computational cells is introduced. The grid may be structured or unstructured, resulting in different implementations of the algorithms and different types of data access patterns.

Most algorithms for solving PDEs could be viewed as an iterative process, where the loop body consists of a (generalized) multiplication of a very large and sparse matrix by a vector containing one or a few entries per cell in the grid. When a structured grid is used, the sparsity pattern of the matrix is pre-determined and highly structured. Only a simple array data structure is needed to store the matrix entries, and the structure is employed in the implementation of the matrix-vector multiplication. The memory access pattern of the codes exhibit large spatial and temporal locality, and the codes are normally very efficient.

For an unstructured grid, the sparsity pattern of the matrix is unstructured and determined at runtime. Normally, the pattern is stored together with the matrix entries using, a sparse storage format such as the compressed sparse row (CSR) format. This leads to indirect addressing in the inner loop of the matrix-vector multiplication, which reduces the efficency [28]. Also, the spatial locality is normally reduced compared to a structured grid discretization because of the more irregular access pattern.

In this paper, we study geographical locality for PDE solvers for problem settings in three space dimensions. The solvers employ both structured and unstructured static grids. We have noted that benchmark codes often solve simplified PDE problems using standardized algorithms, which may result in different performance result than for kernels from advanced application codes. For both solver classes, we therefore perform experiments using a standard benchmark code and a kernel from an industrial application. The two benchmark codes are selected from the NAS NPB3.0-OMP suite [17], where problems of size B are solved. All codes are written in Fortran 90, and parallelized using OpenMP.

Both solvers for structured grids employ a multi-grid iteration. The benchmark code is NAS-MG, which solves the Poisson equation using a finest grid of $256 \times 256 \times 256 = 16.77$ million unknowns. A standard finite element discretization is used, and the solution is computed using eight multi-grid levels. The PDE problem is linear, scalar and has constant coefficients. This reduces the arithmetic work per grid cell, which results in less temporal locality. Also, since the system of equations is linear, all multi-grid levels modifies the same array of solution variables using different strides, which reduces spatial locality. The corresponding industrial code, denoted I-MG, is a kernel for solving the time-independent, compressible Euler equations in three space dimensions. Using a multi-block grid setting, this type of kernel is used for the computation of the airflow around complete aircraft configurations [16]. The Euler equations is a more complex system of five non-linear PDEs, which is discretized using a finite volume method on a curvilinear grid, combined with a flux-vector splitting technique [16]. The number of arithmetic operations per grid cell is much larger than for the benchmark problem, resulting in a high degree of temporal and spatial locality. The solver uses a red-black ordered symmetric Gauss-Seidel-Newton iteration which is embedded into a multi-grid scheme [15]. We perform 10 multi-grid iterations [16] and use five multi-grid levels. In this case, the PDE is non-linear, and the solution at different grid levels must be stored in separate arrays. The finest grid has $128 \times 128 \times 128$ cells, corresponding to 10.49 million unknowns.

Benchmark solvers for unstructured grids are represented by the NAS-CG code. Here, 75 conjugate gradient iterations are performed for a sparse system of equations with an unstructured coefficient matrix. For this synthetic problem, the sparse matrix has a random structure set up in the program. The system of equations has 75000 unknowns, and the sparse matrix has 13708072 non-zero elements, resulting in a non-zero density of 0.24%. As for the NAS MG code, the number of arithmetic operations per grid cell is rather small. The industrial kernel exploiting an unstructured grid is denoted I-CG, and solves the Maxwell equations in electro-magnetics around a fighter jet configuration [10]. Again, the conjugate gradient iteration is used, and the kernel is similar to the one used in the NAS-CG code. The system of equations is solved using 47 iterations. In this case the main difference between the benchmark and the the industrial code lies in the structure and non-zero density of the sparse matrix. The system of equations for the industrial kernel has 1794058 unknowns, and the non-zero density is only 0.0009%. Also, the nature of the unstructured grid used results in a more organized sparse matrix than the random matrix used in the NAS-CG benchmark.

4 Experimental setup

Most of our experiments were performed on a SF15k system. A dedicated domain consisting of four nodes was used, and the scheduling of threads to the nodes was controlled by binding the threads to Solaris processor sets. Each node contains four 900 MHz UltraSPARC-IIICu CPUs and 4 GByte of local memory. The data sets used are all approximately 500 MByte, and are easily stored in a single node. Within a node, the access time to local main memory is uniform. The nodes are connected via a crossbar interconnect, forming a cc-NUMA system. The NUMA-ratio is only approximately 2, which is small compared to other commercial cc-NUMA systems available today.

All application codes were compiled with the Sun ONE Studio 8 compilers using the flags `-fast -openmp -xtarget=ultra3cu -xarch=v9b`, and the experiments were performed using the 12/03-beta release of Solaris 9. Here, a static first-touch page placement strategy is used and support for dynamic, application-initiated migration of data is available in the form of a migrate-on-next-touch feature [25]. Migration is activated using a call to the `madvise(3C)` routine, where the operating system is advised to reset the mapping of virtual to physical addresses for a given range of memory pages, and to redo the first-touch data placement. The effect is that a page will be migrated if a thread in another node performs the next access to it. A similar feature is also available on the Compaq Alpha Server GS-series [1]. On SGI Origin-systems [19], page migration is also available. However, it is implemented using access counters, and no migrate-on-next-touch feature is available. Instead, HPF-style explicit directives for data distribution can be inserted in the code.

We have also used a Sun WildFire system with two nodes for some of our experiments. Here, each node has 16 UltraSPARC-II processors running at 250 MHz. This experimental cc-NUMA computer has CPUs which are of an earlier generation, but includes an interesting dynamic and transparent page placement optimization capability. The system runs a special version of Solaris 2.6, where pages are initially allocated using the first-touch strategy. During program execution a software daemon detects pages which have been placed in the wrong node and migrates them without any involvement from the application code. Furthermore, the system also detects pages which are used by threads in both nodes and replicates them in both nodes. A per-cache-line coherence protocol keeps coherencence between the replicated cache lines.

5 Results

We begin by studying the impact of geographical locality for our codes using the SF15k system. Because of the rather small NUMA-ratio, we can not expect very large differences in performance between different data placement schemes. We focus on isolating the effects of the placement of data, and do not attempt to assess the much more complex issue of the scalability of the codes. The scalability of the NAS benchmarks on the SF15k using an older version of Solaris without memory placement optimization is studied in [11]. First, we measure the execution time for our codes using four threads on a single node. In this case, the first touch policy results in that all application data is allocated locally, and the memory access time is uniform. These timings are denoted UMA in the tables and figures. We then compare the UMA timings to the corresponding execution times when executing the codes in cc-NUMA mode, running a single thread on each of the four nodes. Here, three different data placement schemes are used:

Serial initialization (SI) The main data arrays are initialized in a serial section of the code, resulting in that the pages containing the arrays are allocated on a single node. This is a common situation when application codes are naively parallelized using OpenMP.

Parallel initialization (PI) The main data arrays are initialized in pre-iteration loops within the main parallel region. The first-touch allocation results in that the pages containing the arrays are distributed over the four nodes.

Serial initialization + Migration (SI+MIG) The main arrays are initialized using serial initialization. A migrate-on-next-touch directive is inserted at the first iteration in the algorithm. This results in that the pages containing the arrays will be migrated according to the scheduling of threads used for the main iteration loop.

In the original NAS-CG and NAS-MG benchmarks, parallel pre-iteration loops have been included [17]. The results for PI are thus obtained using the standard codes, while the results for SI are obtained by modifying the codes so that the initialization loops are performed by only one thread. In the I-CG code, the sparse matrix data is read from a file, and it is not possible to include a pre-iteration loop to successfully distribute the data over the nodes using first touch allocation. Hence, no PI results are presented for this code.

In Table 1, the timings for the different codes and data placement settings are shown. The timings are normalized to the UMA case, and execution times in seconds are also given. From the results, it is clear that the geographical locality of data does affect the performance for all four codes. For the I-MG code, both the PI and the SI+MIG strategy are very successful and the performance is effectively the same as for the UMA case. This code has a very good cache hit rate, and the remote accesses produced for the SI strategy do not reduce the performance very much either. For the NAS-MG code the smaller cache hit ratio results in that this code is more sensitive to geographical misplacement of data. Also, NAS-MG contains more synchronization primitives than I-MG, which possibly affects the performance when executing in cc-NUMA mode. Note that even for the NAS-MG code, the SI+MIG scheme is more efficient than PI. This shows that sometimes it is difficult to introduce efficient pre-iteration loops even for structured problems.

Application	UMA	cc-NUMA		
		SI	PI	SI+MIG
NAS-CG	1.00 (233.94s)	1.12 (261.48s)	1.08 (253.31s)	1.04 (243.31s)
NAS-MG	1.00 (20.76)	1.58 (32.75s)	1.43 (29.73s)	1.15 (23.88s)
I-CG	1.00 (39.91s)	1.58 (63.15s)	-	1.15 (46.17s)
I-MG	1.00 (219.05s)	1.18 (258.91s)	1.00 (219.83s)	1.01 (222.22s)

Table 1: Timings for the cc-NUMA settings normalized to the UMA case (timings in seconds in parenthesis)

For the NAS-CG code, the relatively dense matrix results in reasonable cache hit ratio and the effect of geographical misplacement is not very large. Again SI+MIG is more efficient than than PI, even though it is possible to introduce a pre-iteration loop for this unstructured problem. For I-CG, the matrix is much sparser, and the caches are not so well utilized as for NAS-CG. As remarked earlier, it is not possible to include pre-iteration loops in this code. There is a significant difference in performance between the unmodified code (SI) and the version where a migrate-on-next-touch directive is added (SI+MIG).

In the experiments, we have also used the UltraSPARC-III hardware counters [26] and the `cpustat` tool to measure the number of L2 cache misses which are served by local and remote memory respectively. In Table 2, the fraction of remote accesses for the different codes and data placement setting is shown. The results in the SI-column indicates that the assumption that data is allocated in a single node when this scheme is used is correct. Note that the hardware counters measurements shows the results of all activity on the CPUs, including operating system processes and kernel routines. Comparing Tables 2 and 1, it is verified that that the differences

	UMA	cc-NUMA		
		SI	PI	SI + MIG
NAS-CG	0.01%	75.07%	35.93%	6.17%
NAS-MG	0.01%	72.38%	48.53%	11.03%
I-CG	0.01%	67.85%	-	31.39%
I-MG	0.09%	77.11%	4.28%	3.58%

Table 2: Fraction of remote main memory accesses for the applications running with four threads

in overhead between the cc-NUMA cases compared to the UMA timings is related to the fraction of remote memory accesses performed. When running the codes in cc-NUMA mode, remote accesses will always occur due to true communication of data and synchronization. Also, the migrated arrays in the codes could not be aligned to the boundaries of the 8 kByte pages. Hence, distributing these arrays using the PI and SI+MIG schemes, some remote accesses will probably occur because of false-sharing between pages.

We now study the overhead for the dynamic migration in the SI+MIG scheme. In Figures 1(a), 1(b), 2(a), and 2(b), the execution time per iteration for the different codes and data placement settings is shown. As expected, the figures show that the overhead introduced by migration is completely attributed to the first iteration. The time required for migration varies from 0.80 s for the NAS-CG code to 3.09 s for the I-MG code. Unfortunately, we can not measure the number of pages actually migrated, and we do not attempt to explain the differences between the migration times. For the NAS-MG and I-CG codes, the migration overhead is significant compared to the time required for one iteration. If the SI+MIG scheme is used for these codes, approximately five iterations must be performed before there is any gain from migrating the data. For the NAS-CG code the relative overhead is smaller, and migration is beneficial if two iterations are performed. For the I-MG code, the relative overhead from migration is small, and using the SI+MIG scheme even the first iteration is faster than if the data is kept on a single node.

We now examine the scalability of the migration in the SI+MIG scheme. As a measure of the migration overhead we use the difference in execution time between iteration one and two. In Table 3, we present the results obtained when executing the codes using four threads distributed as before, when using 12 threads distributed three-by-three over the nodes, and finally when using 12 distributed threads and large pages (512k instead of 8k). The reason for using large pages is that we expect the overhead to consist of two parts, one for moving the data between the nodes and one for invalidating and recomputing the address translations for the affected pages. The overhead from the first part should depend on the amount of data that is

Application	4 threads	12 threads	12 threads 512K pages
NAS-CG, size B	0.80	0.77	0.16
NAS-MG, size B	2.77	2.52	1.20
I-CG	1.72	1.94	0.60
I-MG	3.09	2.71	1.10

Table 3: Migration overhead for the SI+MIG scheme, measured as the difference in execution time between iteration one and two.

migrated, while the overhead from the second part should depend on the number of pages involved. The results show that the migration overhead is not much affected by using 12 threads instead of four, which is reasonable since the amount of migrated data is about the same. The execution time per iteration for the second and following iterations is not affected significantly by using large pages, implying that the handling of address translations contributes significantly to the total migration overhead. Also, we did not see any substantial false-sharing effects from using larger pages.

Finally, we do a qualitative comparison of the SI+MIG strategy to the transparent, dynamic migration implemented in the Sun WildFire system. In Figures 3(a) and 3(b), we show the results for the I-CG and I-MG codes obtained using 4 threads on each of the two nodes in the WildFire system. Here, the SI+TMIG-curves represent timings obtained when migration and replication is enabled, while the SI-curves are obtained by disabling these optimizations and allocating the data at one of the nodes. Comparing the UMA- and SI-curves in Figures 3(a) and 3(b) to the corresponding curves for SF15k in Figures 2(a) and 2(b), we see that the effect of geographical locality is much larger on WildFire than on SF15k. This is reasonable, since the NUMA-ratio for WildFire is approximately three times larger than for SF15k. From the figures, it is also clear that the transparent migration is active during several iterations. The reason is that, first the software daemon must detect which pages are candidates for migration, and secondly the number of pages migrated per time unit is limited by a parameter in the operating system. One important effect of this is that on the WildFire system, it is beneficial to activate migration even if very few iterations are performed.

6 Conclusions

Our results show that geographical locality is important for the performance of our applications on a modern cc-NUMA system. We also conclude that application-initiated migration leads to better performance than parallel initialization in almost all cases examined, and in some cases the performance is close to that obtained if all threads and their data reside on the same node. Finally, the overhead from the application-initiated migration does not increase when the number of threads per node is increased from 4 to 12, and that this overhead can be reduced using large pages. The main possible limitations of the validity of these results are that the applications involve only sparse, static numerical operators and that the number of nodes and threads used in our experiments are rather small.

Most application programmers ignores (or would preferably ignore) the details of optimizations for cache utilization. Efficient hardware caching schemes and compiler optimizations normally exploits temporal and spatial data locality to a large degree, which results in reasonable and robust performance for most applications. In the same way, it is desirable that the optimization techniques for improving geographical locality should be highly transparent. Ideally, the programmer should not have to explicitly influence the placement of data by introducing modifications in the code, and the performance of the application should be reasonably good and robust with respect to changes of algorithm, data set and computer system. The standard OpenMP programming model has no constructs for affecting the placement of data in a cc-NUMA system, and following the argument above such additions should if possible be avoided.

The application-initiated migration used as in this study is not transparent, instead the programmer has to invoke the migration explicitly when the data access pattern changes. In well-written OpenMP codes, the number of parallel regions is often small, and each such region often implements a part of the algorithm where a given data access pattern is used. Hence, the start of a new parallel region is an indicator of that the memory access pattern might change. By automatically or implicitly introducing a migrate-on-next-touch directive for all shared data at the beginning of a parallel region, the performance benefits of the migration scheme can be exploited in a transparent way without changing the OpenMP syntax or violating the design goals of OpenMP. The major case against such a strategy could be the overhead introduced by the migration, especially if the applications exhibit several and small parallel regions. Further studies have to be made for a large class of OpenMP applications to investigate the benefits and drawbacks of such an implicit migrate-on-next-touch strategy before we can draw any conclusions about the general applicability.

Finally, we have also performed a qualitative comparison of the results for the commercial cc-NUMA to results obtained on a prototype cc-NUMA system, a Sun WildFire server. This system supports fully transparent adaptive memory placement optimization in the hardware, and our results show that this is also a viable alternative on cc-NUMA systems. In fact, for applications where the access pattern changes dynamically but slowly during execution, a self-optimizing system is probably the only viable solution for improving geographical locality.

References

[1] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA machines. *Scientific Programming*, 8:163–181, 2000.

[2] Timothy Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, Sept 1993.

[3] J. Mark Bull and Chris Johnson. Data Distribution, Migration and Replication on a cc-NUMA Architecture. In *Proceedings of the Fourth European Workshop on OpenMP*. `http://www.caspur.it/ewomp2002/`, 2002.

[4] Rohit Chandra, Ding-Kai Chen, Robert Cox, Dror E. Maydan, Nenad Nedeljkovic, and Jennifer M. Anderson. Data distribution support on distributed shared memory multiprocessors. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 334–345. ACM Press, 1997.

[5] Rohit Chandra, Scott Devine, Ben Verghese, Anoop Gupta, and Mendel Rosenblum. Scheduling and page migration for multiprocessor compute servers. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 12–24. ACM Press, 1994.

[6] Alan Charlesworth. The sun fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 7–7. ACM Press, 2001.

[7] Julita Corbalan, Xavier Martorell, and Jesus Labarta. Evaluation of the memory page migration influence in the system performance: the case of the sgi o2000. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129. ACM Press, 2003.

[8] D. Culler, J.P Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach.* Morgan Kaufman, 1998.

[9] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry-Standard API for Shared-Memory Programming. *Computational Science and Engineering, IEEE*, 5(1):46–55, Jan.-March 1998.

[10] Fredrik Edelvik. *Hybrid Solvers fo the Maxwell Equations in Time-Domain.* Doctoral thesis, Mathematics and Computer Science, De-

partment of Information Technology, University of Uppsala, may 2002. `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-2156`.

[11] Nathan R. Fredrickson, Ahmad Afsahi, and Ying Qian. Performance characteristics of openmp constructs, and application benchmarks on a large symmetric multiprocessor. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 140–149. ACM Press, 2003.

[12] Kourosh Gharachorloo, Madhu Sharma, Simon Steely, and Stephen Van Doren. Architecture and design of alphaserver gs320. *ACM SIGPLAN Notices*, 35(11):13–24, 2000.

[13] Erik Hagersten and M. Koster. WildFire: A Scalable Path for SMPs. In *Proceedings of the 5th International Symposium on High-Performance Architecture*, 1999.

[14] John L. Hennessy and David A. Patterson. *Computer Architecture;A Quantative Approach*. Morgan Kaufman, 3rd edition, 2003.

[15] A. Jameson. Solution of the euler equations for two-dimensional, transonic flow by a multigrid method. *Appl. Math. Comp.*, 13:327–356, 1983.

[16] A. Jameson and D. A. Caughey. How many steps are required to solve the euler equations of steady, compressible flow: In search of a fast solution algorithm. In *15th Computational Fluid Dynamics Conference*, 2001.

[17] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center, 1999.

[18] Kevin Krewell. EV7 Stresses Memory Bandwidth. *Microprocessor Report*, March 24 2003.

[19] James Laudon and Daniel Lenoski. The SGI Origin: a ccNUMA highly scalable server. In *Proceedings of the 24th annual international symposium on Computer architecture*, pages 241–251. ACM Press, 1997.

[20] Timothy G. Mattson. How good is OpenMP. *Scientific Programming*, 11:81–93, 2003.

[21] Dimitrios S. Nikolopoulos, Constantine D. Polychronopoulos, and Eduard Ayguadi. Scaling irregular parallel codes with minimal programming effort. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 16–16. ACM Press, 2001.

[22] Dimitros S. Nikolopoulos, Theodore S. Papatheodorou, Constantine D. Polychronopoulos, Jesus Labarta, and Eduard Ayguade. A transparent runtime data distribution engine for OpenMP. *Scientific Programming*, 8:143–162, 2000.

[23] Lisa Noordergraaf and Ruud van der Pas. Performance experiences on Sun's Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 38. ACM Press, 1999.

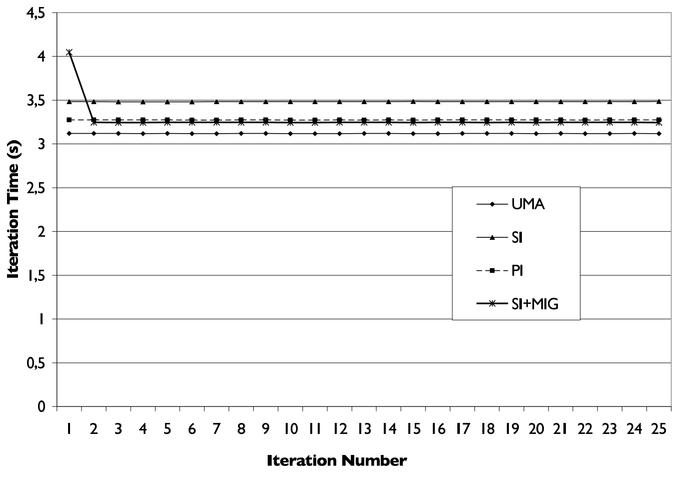
[24] Silicon Graphics. *SGI Origin 3000 series*, 2000.

[25] Sun Microsystems, `http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf`. *Solaris Memory Placement Optimization and Sun Fire servers*, January 2003.

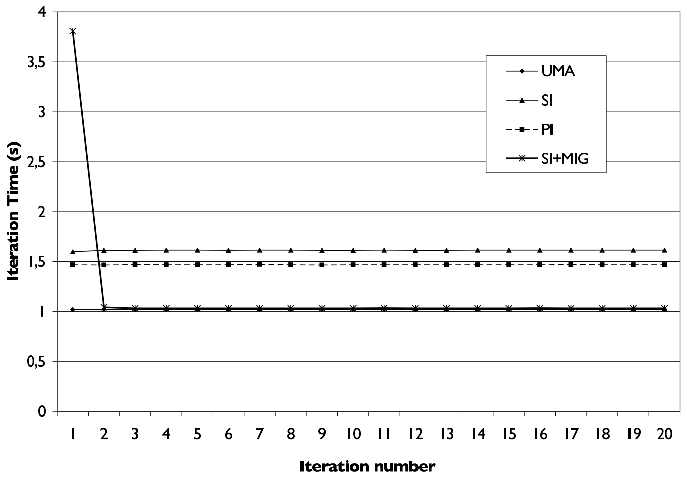
[26] Sun Microsystems, `http://www.sun.com/processors/manuals`. *UltraSPARC III Cu User's Manual*, 2003.

[27] Ben Verghese, Scott Devine, Anoop Gupta, and Mendel Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279–289. ACM Press, 1996.

[28] Richard Vuduc, James W. Demmel, Katherine A. Yelick, Shoaib Kamil, Rajesh Nishtala, and Benjamin Lee. Performance optimizations and bounds for sparse matrix-vector multiply. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, pages 1–35. IEEE Computer Society Press, 2002.

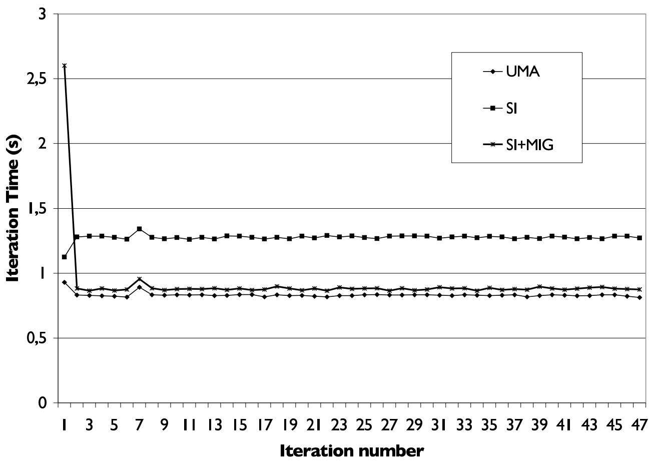


(a) Execution time per iteration of NAS-CG. Only the first 25 iterations are shown in the graph.

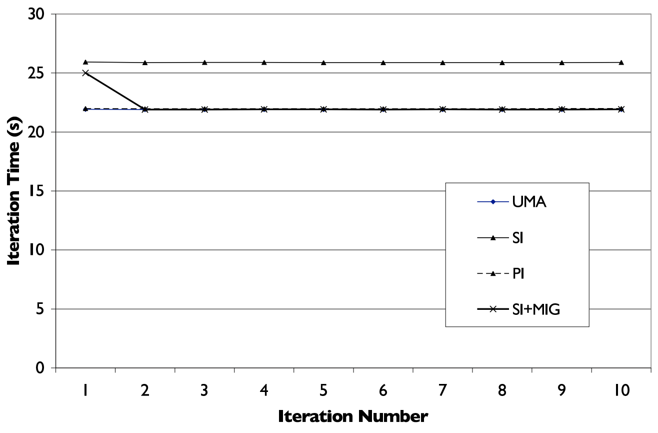


(b) Execution time per iteration of NAS-MG.

Figure 1: Execution time per iteration for NAS-CG and NAS-MG on the SF15K using 4 threads.

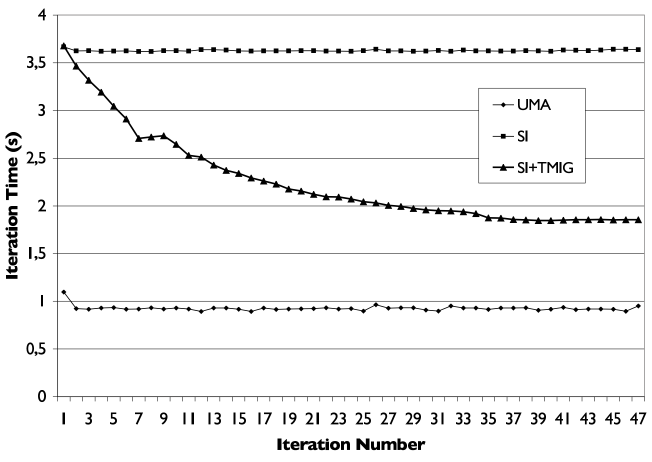


(a) Execution time per iteration of I-CG.

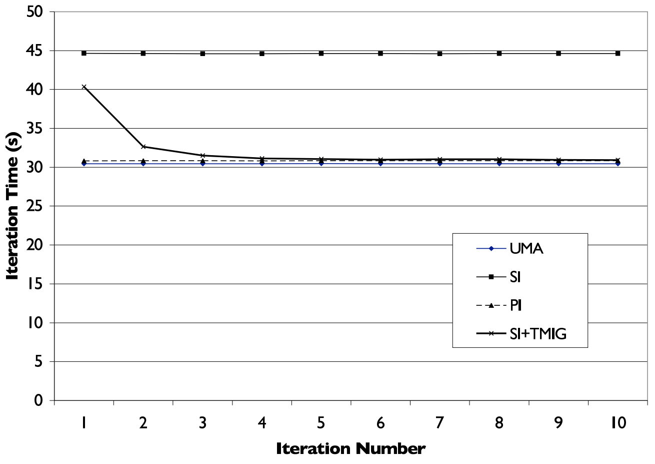


(b) Execution time per iteration of I-MG.

Figure 2: Execution time per iteration for I-CG and I-MG on the SF15K using 4 threads.



(a) Execution time per iteration of I-CG.



(b) Execution time per iteration of I-MG.

Figure 3: Execution time per iteration for I-CG and I-MG on the Sun Wild-Fire using 8 threads.