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 one aspect of the non-uniformity is the physical distance between the cc-NUMA nodes. We compare the well established first-touch strategy to an application-initiated page migration strategy as means of increasing the geographical locality for a set of important scientific applications. 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.

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, 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. 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 is an active research area, see e.g. [1,2,3,4].

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:

M. Bubak et al. (Eds.): ICCS 2004, LNCS 3037, pp. 9–16, 2004.

- 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?

Most experiments presented in this paper are performed using a Sun Fire 15000 (SF15k) system, which is a commercial cc-NUMA computer. Some experiments are also performed using a Sun WildFire prototype system [5].

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 [2], implicitly invoked by software [6], or they may be implicitly invoked by the computer system [7,8,9].

2 Applications

To evaluate different methods for improving geographical locality we study the performance of four solvers for large-scale partial differential equation (PDE) problems. 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. 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. Here, the spatial locality is normally reduced compared to a structured grid discretization because of the more irregular access pattern.

We have noted that benchmark codes often solve simplified PDE problems using standardized algorithms, which may lead to different performance results

than for kernels from advanced application codes. We therefore perform experiments using kernels from industrial applications as well as standard benchmark codes from the NAS NPB3.0-OMP suite [10]. More details on the applications are given in [11]. All codes are written in Fortran 90, and parallelized using OpenMP. The following PDE solvers are studied:

NAS-MG. The NAS MG benchmark, size B. Solves the Poisson equation on a $256 \times 256 \times 256$ grid using a multi-grid method.

I-MG. An industrial CFD solver kernel. Solves the time-independent Euler equations describing compressible flow using an advanced discretization on a grid with $128 \times 128 \times 128$ cells. Also here a multi-grid method is used.

NAS-CG. The NAS CG benchmark, size B. Solves a sparse system of equations with an unstructured coefficient matrix using the conjugate gradient method. 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%.

I-CG. An industrial CEM solver. Solves a system of equations with an unstructured coefficient matrix arising in the solution of the Maxwell equations around an aircraft. Again, the conjugate gradient method is used. This system of equations has 1794058 unknowns, and the non-zero density is only 0.0009%.

3 Results

On the 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 [12]. 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.

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 coherence between the replicated cache lines.

We begin by studying the impact of geographical locality for our codes using the SF15k system. We focus on isolating the effects of the placement of data, and do not attempt to assess the more complex issue of the scalability of the codes. 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.

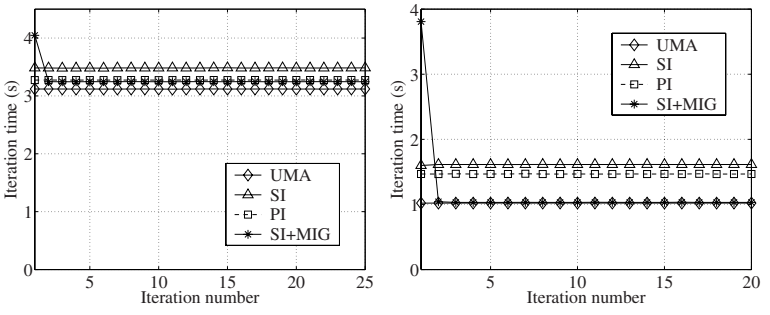
Serial initialization + Migration (SI+MIG). The main arrays are initialized using serial initialization. A migrate-on-next-touch directive is inserted at the first iteration in the algorithm. This results in that the pages containing the arrays will be migrated according to the scheduling of threads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

In the original NAS-CG and NAS-MG benchmarks, parallel pre-iteration loops have been included [10]. 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, where the times are given also in seconds. 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,

Table 1. Timings and fraction of remote memory accesses for the different codes and data placement settings. The timings for the cc-NUMA settings are normalized to the UMA case

Application	Time				Remote accesses			
	UMA	SI	PI	SI+MIG	UMA	SI	PI	SI+MIG
NAS-CG	1.00 (233.9s)	1.12	1.08	1.04	0.0%	75.1%	35.9%	6.2%
NAS-MG	1.00 (20.8s)	1.58	1.43	1.15	0.0%	72.4%	48.5%	11.0%
I-CG	1.00 (39.9s)	1.58	N/A	1.15	0.0%	67.9%	N/A	31.4%
I-MG	1.00 (219.1s)	1.18	1.00	1.01	0.1%	77.1%	4.3%	3.6%



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

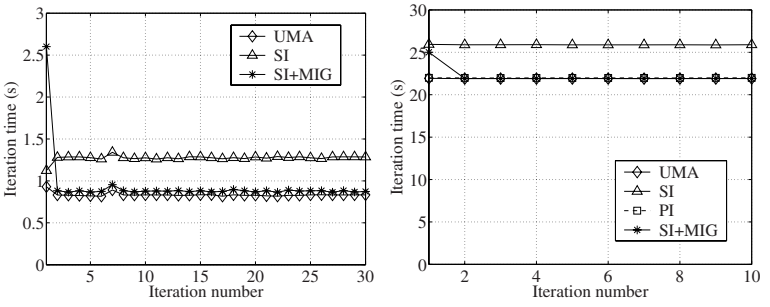
(b) Execution time per iteration of NAS-MG

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

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 also for structured problems.

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 to measure the number of L2 cache misses which are served by local and remote



(a) Execution time per iteration of I-CG

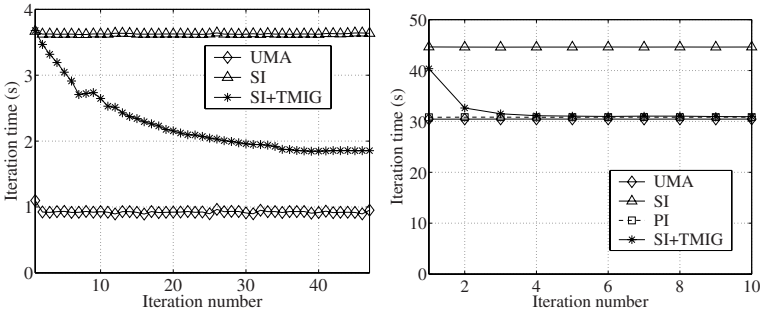
(b) Execution time per iteration of I-MG

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

memory respectively. In Table 1, the fractions of remote accesses for the different codes and data placement settings are shown. Comparing the different columns of Table 1, it is verified that that the differences in overhead between the cc-NUMA cases compared to the UMA timings is related to the fraction of remote memory accesses performed.

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. A study of the scalability of the SI+MIG scheme is performed in [11].

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



(a) Execution time per iteration of I-CG

(b) Execution time per iteration of I-MG

Fig. 3. Execution time per iteration for I-CG and I-MG on the Sun WildFire using 8 threads

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.

4 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. 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.

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 ac-

cess pattern changes dynamically but slowly during execution, a self-optimizing system is probably the only viable solution for improving geographical locality.

References

1. Noordergraaf, L., van der Pas, R.: Performance experiences on Sun's Wildfire prototype. In: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM), ACM Press (1999) 38
2. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C.A., Offner, C.D.: Extending OpenMP for NUMA machines. Scientific Programming **8** (2000) 163–181
3. Nikolopoulos, D.S., Papatheodorou, T.S., Polychronopoulos, C.D., Labarta, J., Ayguade, E.: A transparent runtime data distribution engine for OpenMP. Scientific Programming **8** (2000) 143–162
4. Bull, J.M., Johnson, C.: 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)
5. Hagersten, E., Koster, M.: WildFire: A Scalable Path for SMPs. In: Proceedings of the 5th International Symposium on High-Performance Architecture. (1999)
6. Nikolopoulos, D.S., Polychronopoulos, C.D., Ayguadi, E.: Scaling irregular parallel codes with minimal programming effort. In: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM), ACM Press (2001) 16–16
7. Verghese, B., Devine, S., Gupta, A., Rosenblum, M.: 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, ACM Press (1996) 279–289
8. Chandra, R., Devine, S., Verghese, B., Gupta, A., Rosenblum, M.: Scheduling and page migration for multiprocessor compute servers. In: Proceedings of the sixth international conference on Architectural support for programming languages and operating systems, ACM Press (1994) 12–24
9. Corbalan, J., Martorell, X., Labarta, J.: 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, ACM Press (2003) 121–129
10. Jin, H., Frumkin, M., Yan, J.: The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center (1999)
11. Löf, H., Nordén, M., Holmgren, S.: Improving geographical locality of data for shared memory implementations of pde solvers. Technical Report 006, Department of Information Technology, Uppsala University (2004)
12. Sun Microsystems `http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf`: Solaris Memory Placement Optimization and Sun Fire servers. (2003)