Geographical Locality and Dynamic Data Migration for OpenMP Implementations of Adaptive PDE Solvers

Markus Nordén, Henrik Löf, Jarmo Rantakokko, and Sverker Holmgren

Uppsala University, Department of Information Technology
Box 337, 751 05 Uppsala, SWEDEN
markus.norden@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*. In this article, we study the performance of a parallel PDE solver with adaptive mesh refinement. The solver is parallelized using OpenMP and the adaptive mesh refinement makes dynamic load balancing necessary. Due to the dynamically changing memory access pattern caused by the runtime adaption, it is a challenging task to achieve a high degree of geographical locality.

The main conclusions of the study are: (1) that geographical locality is very important for the performance of the solver, (2) that the performance can be improved significantly using dynamic page migration of misplaced data, (3) that a migrate-on-next-touch directive works well whereas the first-touch strategy is less advantageous for programs exhibiting a dynamically changing memory access patterns, and (4) that the overhead for such migration is low compared to the total execution time.

1 Introduction

Today, most parallel solvers for large-scale PDE applications are implemented using a local address space programming model such as MPI. During the last decade there has also been an intensified interest in using shared address space programming models like OpenMP for these type of applications. A main reason is that an increasing number of applications require the use of adaptive mesh refinement (AMR), and in this case the work and data need to be dynamically repartitioned at runtime to get good parallel performance. Using a local address space model, an extensive programming effort is needed to develop parallel PDE solver implementations that include such mechanisms. Using a shared address space model, the programming effort for producing a working parallel code can be reduced significantly. Another driving force for the use of shared address space models is the recent development in computer architecture; Emerging computer systems are built using multi-threaded and/or multi-core processors, future standard computational nodes will comprise an increasing number of threads that share a single address space. Codes using a programming model like OpenMP can then be transparently and easily used on different size systems, ranging from laptops with a single multi-threaded CPU to large shared memory systems with many such CPUs.

Most larger shared memory computers are built from nodes (chips) with one or several processors (cores), forming a cache-coherent non-uniform memory architecture (cc-NUMA). In a NUMA system, the latency for a main memory access depends on whether data is accessed at a local memory location or at a remote location. One characteristic property of this type of computer system is the *NUMA-ratio*, which is defined as the quotient of the remote and local access times. The non-uniform memory access time leads to that the *geographical locality* of data potentially affects the application performance. Here, optimal geographical locality corresponds to that the data is distributed over the nodes in a way that matches with the thread accesses in the best possible way. Good geographical locality can be achieved by carefully selecting the node where data is allocated at initiation, and/or by introducing some form of dynamic migration of data between the nodes during execution [1, 2, 3, 4].

A main reason for the complexity of local address space implementations of AMR PDE solvers is that the programmer *must* explicitly control and modify the partitioning of work and data during execution. If suitable algorithms for partitioning and load balancing are used and the migration of data is efficiently implemented, a local address space implementation will regularly exhibit good parallel performance. In a shared address space model, the native work sharing constructs and transparent communication result in that it is much less demanding to develop a working parallel code. However, the aspects of work partitioning and load balancing must normally still be considered to obtain robust and competitive parallel performance. In programming models like OpenMP, the work partitioning and load balancing can easily be performed using the same well-developed and efficient algorithms as for local address space models, resulting in that the potential for good parallel efficiency is retained. Data distribution is not considered in OpenMP, and poor geographical locality could possibly lead to deteriorated performance. In this paper, we study the implementation of a structured adaptive mesh refinement (SAMR) PDE solver and attempt to answer the following questions:

– How large is the impact of geographical locality on the performance?
– Can the performance be improved through dynamic migration of misplaced data?
– How large is the migration overhead?

The rest of the paper is organized as follows: In Section 2 we describe existing parallel SAMR solvers and techniques used for distribution of work and data. In Section 3 we consider the model PDE application that is solved, in Section 4 we introduce the NUMA computer system used, and in Section 5 we give some details about the implementation and experimental setup. In Section 6 we present performance results, and in Section 7 we conclude.

2 Parallel SAMR Solvers

Most existing parallel implementations of large-scale SAMR PDE solvers [5], e.g., AMROC [6], PARAMESH [7], GrACE [8], and SAMRAI [9] exploit a local address space model implemented using MPI. The parallelization is based on grid blocks and each MPI process is responsible for the computations corresponding to one or more blocks or parts of blocks. For easier balancing of the computational work, large blocks are often first split into smaller blocks. All blocks are then assigned to the processors with a load balancing algorithm. In the computations, a small amount of communication is always needed for interpolating grid function values between some blocks in different processes.

Both patch-based and domain-based approaches for dynamic work partitioning and load balancing in local address space SAMR solvers have been developed. In patch-based methods the blocks at one level are partitioned over all processes while in the domain-based the computational domain is partitioned and the partitions are projected to the different grid levels. Hybrid versions combining patch-based and domain based methods have also been considered [10]. Furthermore, the algorithms can be categorized into scratch-remap and diffusion type. Using a scratch-remap strategy, a new partitioning is computed without considering previous partitionings, but the new partitioning is re-mapped according to previous data distribution in order to minimize data migration. In diffusion algorithms the partitioning is computed with a previous partitioning as a starting point. Scratch-remap strategies tries to optimize load balance and communication while the main objective in diffusion algorithms is to minimize data migration with load balance and communication as secondary objectives [11]. So far, there is no single algorithm that performs best for all types of applications or not even for all states of a particular application, see e.g. [10]. General dynamic load balancing algorithms for SAMR solvers remains an open field of research. In hierarchical AMR methods a common choice is to use space filling curves for clustering blocks into partitions [6, 7, 8, 9, 12], using Morton or Hilbert ordering. Space filling curves are fast and offer both locality within and between levels in the grid hierarchy. For flat unstructured AMR methods, graph partitioning methods are more common [11, 13, 14] and have better locality properties than the space filling curves. For flat, SAMR applications and multi-block grids the graph partitioning algorithms are also preferable [15, 16].

For the experiments performed in this paper, it is important to minimize the data migration in addition to achieve a good load balance and minimize communication. A diffusion partitioning algorithm then becomes the natural choice, and we have chosen to us an algorithm of this type from the Jostle package [14].

In [17], a shared memory parallelization of a SAMR solver is presented. The code is parallelized using OpenMP and the experiments are performed on an SGI Origin system. A reasonable amount of geographical data locality is achieved by using SGI Origin's first touch data placement policy, i.e., data is allocated in the local memory of the thread first touching a grid block. In [12] a shared address space parallelization using POSIX-multi-threading is discussed. Here, explicit localization of data is implemented by using private memory in the threads for storing the blocks, i.e., each thread has access to the grid hierarchy but stores only its blocks in private data structures in local memory. Moreover, to guarantee geographical locality the threads are explicitly bind to single CPU's.

In [18] different programming models using MPI, OpenMP and hybrid MPI-OpenMP for parallelizing a SAMR solver are compared on a Sun Fire 15K, which is also a NUMA system. Parallelization is performed both at block level and at loop level. It is shown that the coarse grain block level parallelization with MPI gives the best performance as long as the number of blocks is large enough for a good load balance, otherwise a mixed MPI-OpenMP model is better due to better distribution of the work. The standard OpenMP implementation suffers from poor geographical data locality and does not perform as well as the corresponding MPI implementations.

3 The PDE solver

As a representative model problem we solve the advection equation

$$u_t = u_x + u_y$$

with periodic boundary conditions on a square. The initial solution is a Gaussian pulse. As time evolves the pulse moves diagonally out through one of the corners of the domain and comes back in from the opposite corner without changing shape. The PDE is discretized by a second-order accurate finite difference method in space and the classical fourth order Runge-Kutta method in time. We use a structured cartesian grid and divide the domain into a fixed user-defined number of blocks. As a simple error estimate in the adaption criterion we use the maximum value of the solution in a block. In a real-life application, a more sophisticated error estimate e.g. based on applying the spatial difference operator on a coarse and a fine block discretization would be used [19]. However, this would not affect the parallel performance much, and the conclusions drawn from the experiments presented later will not change. If the error estimate of a block exceeds a threshold, the resolution of the grid is refined with a factor two in the entire block. On the contrary, if the error is small enough, the grid in the block is coarsened with a factor two.

The code is written in Fortran 90 and parallelized using OpenMP. The parallelization is coarse grained over entire blocks, i.e. each thread is responsible for a set of blocks. The blocks have two layers of ghost cells which are updated by reading data from the neighboring blocks. When the grid resolution changes in any of the blocks, the entire grid block structure is repartitioned using the Jostle diffusion algorithm and the work partitioning between the threads is changed accordingly.

Before the main time-evolution loop starts, the solution is initialized. This is done in parallel, according to an initial partitioning that was defined when the grid was created. After this the error is estimated and the grid adapted if necessary. This procedure, initialization and adaption, is repeated until the error estimates are satisfied in all blocks. Thereafter, the grid is repartitioned and the main computations starts. The computational kernel of our SAMR application is presented in pseudocode in Figure 1. In the code, the procedure `Diff()` performs the necessary interpolation between grid blocks and applies the spatial difference operator for all blocks in the grid. In the experiments presented

```
do t=1,Nt
   if (t mod adaptInterval=0) then
      Estimate error per block.
      Adapt blocks with inappropriate resolution.
      Repartition the grid.
      Migrate blocks (if migration is activated).
   end if
   F1=Diff(u);
   F2=Diff(u+k/2*F1)
   F3=Diff(u+k/2*F2)
   F4=Diff(u+k*F3)
   u=u+k/6*F1+k/3*F2+k/3*F3+k/6*F4
end do
```

Fig. 1. Pseudocode for the computational kernel of our SAMR application.

later, we perform a total of 20000 time steps. Adaption, partitioning and migration (if active) is performed every `AdaptInterval` time step, where we use `AdaptInterval`=20. We use a discretization with 16x16 blocks, and the adaption criterion results in three different block sizes: 100x100, 200x200 and 400x400. When a block is refined or coarsened new memory is allocated and the old block is discarded. At a typical iteration the resident working set was about 350 MB. We define the load balance γ by

$$\gamma = \frac{\max_i p_i}{\frac{1}{n}\sum_{i=0}^{n} p_i}, \tag{1}$$

where p_i is the amount of work in partition i and n the number of partitions. Using $n = 4$ and sampling the load balance γ_j after each partitioning, we got an arithmetical mean of 1.09 with a standard deviation 0.12. Hence, the diffusion type partitioner used gives a reasonably good load balance.

4 The NUMA system

All experiments presented below were performed on a Sun Fire 15000 system, where a dedicated domain consisting of four nodes was used. Each node contains four 900 MHz UltraSPARC-IIICu CPUs and 4 GByte of local memory, and each CPU has an off-chip 8MB L2 cache. 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 with NUMA-ratio approximately 2.0.

The codes was compiled with the Sun STUDIO 11 compiler using the flags `-fast -xarch=v8plusa -xchip=ultra3cu`, and the experiments were performed using the 4/04 release of Solaris 9. When an application starts the Solaris scheduler assigns each thread a home node (called *locality group* or *lgroup* in Solaris terminology). Although threads are allowed to execute on any node the scheduler tries to keep the threads to their home node. By default, memory is allocated according to a first-touch strategy which, in an ideal case, means that memory will be allocated to the home node to create good geographical locality.

In Solaris, dynamic migration of pages between nodes can be performed using a directive with a migrate-on-next-touch semantic using the `madvise(3C)` library call [20]. The directive tags pages for migration and the kernel resets the address translation for these pages. Since the TLB is handled by software in Solaris, dirty translations needs to be invalidated by a TLB shoot down procedure for all CPUs that have executed the address space. After the shot down the pages have no physical address associated with them. When a thread accesses one of these pages a minor page fault occurs and the contents of the page is migrated, i.e. physically copied, to a new page allocated in the node where the faulting thread executes. If the new page is physically allocated to the node where the contents resides, there is a fast-path, no data is actually copied. The overhead of the migration can be divided into two parts: the overhead from TLB shoot down and the cost of copying data. The shoot down overhead is dependent on how many pages are shot down and for how many CPU:s. Due to kernel consistency issues this procedure needs to be serialized using global locking, see Teller [21].

A migrate-on-next-touch directive is also available on the Compaq Alpha Server GS-series [22]. On SGI Origin-systems [23], dynamic 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. Tikir et al [24] showed that a migrate-on-next-touch directive can be used to create a transparent data distribution engine based on hardware access counts. Also, Spiegel et. al [25] showed how to use the migrate-on-next-touch call to speed up an hybrid CFD solver.

5 Experimental Methodology and Setup

In the SAMR application studied here, the data distribution corresponding to the initial first-touch allocation will not be optimal since we need to maintain a good load balance. The partitioner will in many cases assign blocks where some or all of them were initially allocated on a node different from the home node of a given thread. Also, in our implementation, when a block is refined or coarsened, new memory is allocated and the old block is discarded. As the adaption phase precedes the partitioner, new blocks might be allocated by the first-touch strategy to a remote node depending on the outcome of the partitioner.

To increase geographical locality we can use page migration to migrate the data of each partition to the home locality group (node) of the corresponding thread. The simplest strategy would be to migrate all blocks after each partitioning. However, if the partitioner has a notion of locality, such as a diffusion partitioner, the number of blocks that change partition will be lower than the total amount of blocks in each partition. As a consequence we keep track of inter-partition block movements and only migrate the blocks that change partition after each partitioning step. Since the migration is driven by page faults we need to be careful how we touch the data. Each block has a layer of ghost cells to simplify the interpolation between blocks of different levels of refinement. The ghost cells are normally accessed first in a communication step which means that these pages will be migrated to a neighboring thread if a migrate-on-next-touch directive is used. To avoid this behavior we added a sequence of code where the thread touches the entire block, including the ghost cells directly after the migrate-on-next-touch call.

5.1 Experimental Setups

To investigate the performance impact of geographical locality we have performed experiments where the application was executed on using four threads on the following four configurations

UMA All threads confined to the same node. Migration not active.
NUMA One thread per node. Migration not active.
NUMA-MIG One thread per node. Migrate data belonging to blocks that are transfered to another node. Force immediate migration by touching pages.

To make sure that threads stay in their home nodes we used the `SUNW_MP_PROCBIND` environment variable to bind each thread to a specific CPU. We also kept the system unloaded apart from the application studied. In the UMA case, all accesses will be local. However, there is a risk that the performance of the code will be inhibited by the limited bandwidth provided by a single node. In the NUMA cases the aggregate bandwidth to main memory is four times higher. We align data to page boundaries by interposing the Fortran 90/95 `allocate()` routine. Since the SAMR application allocates new blocks in parallel we used the `mtmalloc` allocator. This allocator is part of Solaris and it is much more scalable than the standard allocator. We mapped all allocations to the `valloc()` routine of mtmalloc. This will result in that the smallest possible block of data is a memory page. The memory waste was found to be very low. In total, the application allocated 241540 8kB pages which is close to 2 GB of data in 7636 calls to `allocate`. The waste due to alignment was about 40 MB.

To quantify the effect of geographical locality we measure the number of remote accesses generating from the CPUs using the UltraSPARC-IIICu hardware counters. We define the number of remote accesses as the difference between the total amount of local L2-cache accesses (`EC_miss_local`) and the total amount of L2-cache misses (`EC_misses`). The hardware counter data was sampled using the Sun Performance Analyzer. To reduce the file size of the hardware counter sampling only 4000 times steps of the 20000 were executed. We believe that the basic miss ratio characteristics can still be observed using only a subset of the iterations. Furthermore, the Solaris kernel (kstat) provides counters for the amount of pages migrated to and from a node and the Solaris tool trapstat was used to sample the amount of time spent handling address translations.

6 Results

Table 1 shows both total execution time and hardware counter data from the three setups. We can see from Table 1 that the NUMA case runs slower than the UMA case and the NUMA-MIG case. The number of remote accesses is also

	UMA	NUMA	NUMA-MIG
Total Execution Time	4.09 h	6.64 h	3.99 h
L2 Miss Ratio	4.3%	3.9%	4.2%
L2 Remote Ratio	0.2%	62.9%	8.1%

Table 1. Execution time measurements and hardware counter data from the three different experimental setups

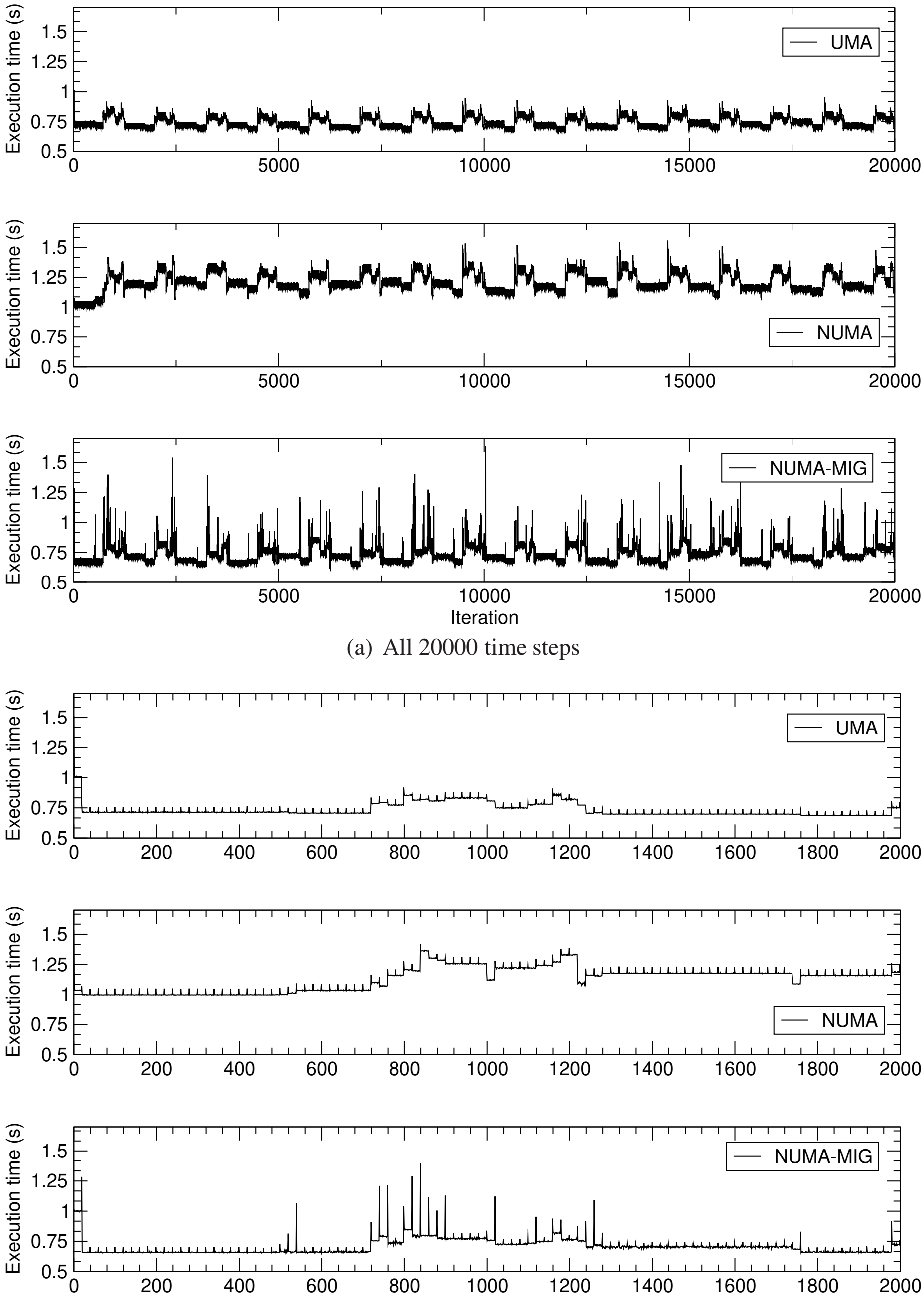
much higher for this case compared to the UMA and NUMA-MIG cases which shows that the NUMA case exhibits a low degree of geographical locality. It is also clear that the effect of page migration is large since the amount of remote accesses for the NUMA-MIG case is much lower compared to the NUMA case. Remember that we can not completely remove all remote accesses since the usage of ghost cells will result in a small amount of communication.

Figure 2 shows the entire execution for the UMA, NUMA and NUMA-MIG cases. To be able to compare performance we have aligned the graphs vertically and all three graphs have the same scale. It is clear that the impact of geographical locality is significant even though the NUMA-ratio of the SF15K is only about two. By comparing the execution time of UMA and NUMA-MIG we see that we can increase the geographical locality using a migrate-on-next-touch directive. Surprisingly, the execution time for the NUMA-MIG case is lower than the UMA case. This can be explained by the fact that in the UMA case all CPU:s of the node will be used resulting in a very high memory pressure on that node. In the NUMA cases each thread will have the entire node for itself resulting in a higher aggregate bandwidth for the application to use.

	Migrated To	Migrated From	Total Net Flow	Total Traffic
Thread 0	346479	356903	-10424	703382 (2.9 GB)
Thread 1	318407	319417	-1010	637824 (5.4 GB)
Thread 2	249414	243203	6211	492617 (3.8 GB)
Thread 3	192122	186899	5223	379021 (3.4 GB)

Table 2. Migration statistics for the NUMA-MIG case collected using Solaris kernel statistics (kstat). Column 4 shows the net data flow from a node where the thread executed. A negative value indicates that more pages were migrated from the node. Column 5 shows the sum of columns 2 and 3 ie the total amount of migrations for one node. The total amount of migrated pages for all nodes was 2212844 (16.88 GB)

Table 2 shows migration statistics for the application in the NUMA-MIG case. At a typical iteration the resident working set was about 350 MB which



(a) All 20000 time steps

(b) First 2000 time steps

Fig. 2. The impact of geographical locality on performance. Migration, adaption and partitioning is triggered every 20:th time step.

corresponds to 44800 8kB pages. Migration was triggered every 20:th time steps which gives a total of 1000 times. If all the data migrate at each migration this would correspond to a total traffic of 44.8 million pages. Comparing this figure to the total amount of pages migrated (2.21 million) we can conclude that a small fraction (5%) of all the pages are migrated. Assuming that the migrations are evenly distributed over time, only 2213 pages (18.1 MB) are migrated at each migration. Hence, we conclude that the amount of data migrated is fairly low. This together with the fact that the NUMA-MIG case executes faster than the UMA case indicates that the overheads of migration are low for the experiments performed. Using the trapstat tool we found that the solver (all cases) spends at most 1.0% of its time (2.4 mins) in the Solaris page fault trap handler. This fact further supports the conclusion that the overheads from migration are low.

7 Conclusions

In this paper we have investigated the impact of geographical locality for an adaptive PDE solver. This application has a dynamic access pattern which implies that a system needs to support some kind of runtime data distribution to minimize the effects of geographical locality. Our results show that the impact of geographical locality is large even though the NUMA-ratio of the system used is only two. We also show that we can significantly improve geographical locality and overall performance using a library call with a migrate-on-next-touch semantic.

The overheads of migration was found to be low which can be attributed to two facts. First, our experiments were performed using only four threads. The overheads from page migration will probably increase with the number of nodes and CPUs. Second, for SAMR to be efficient the refined area of the mesh needs to be rather small. This indicates that the amount of data that needs to be migrated will be low. The refinement patterns of AMR solvers often vary a lot depending on the physics of the problem studied. If data migration is to be used in a more general setting the frequency of invoking a migrate-on-next-touch call must be tuned to match the refinement patterns of the studied problem. If large amounts of data needs to be migrated we may have to reduce the number of migrate calls to amortize the overhead over several time steps. However, for the model problem studied in this article, using a diffusion type partitioner resulted in fairly low amounts of data migrations.

We believe that a call or directive with a migrate-on-next-touch semantic can be a useful addition to an architecture-independent language like OpenMP. Since such a directive is invoked by the programmer we do not have to spend system resources monitoring geographical locality were thread-data affinity is not critical for performance. Furthermore if a system support a transparent mechanism for increasing geographical locality, a migrate-on-next-touch directive could serve as a useful hint to the system.

Bibliography

[1] Wilson, K.M., Aglietti, B.B.: Dynamic page placement to improve locality in CC-NUMA multiprocessors for TPC-C. In: Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing, New York, NY, USA, ACM Press (2001) 33–33

[2] 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

[3] Holmgren, S., Nordén, M., Rantakokko, J., Wallin, D.: Performance of PDE Solvers on a Self-Optimizing NUMA Architecture. Parallel Algorithms and Applications **17**(4) (2002) 285–299

[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] Rendleman, C.A.: Parallelization of structured, hiearchical adaptive mesh refinement algorithms. Computing and Visualization in Science **3** (2000) 147–157

[6] Deiterding, R.: Construction and application of an amr algorithm for distributed memory computers. In: Adaptive Mesh Refinement – Theory and Applications, Proc. of the Chicago Workshop on Adaptive Mesh Refinement Methods, Springer (2003) 361–372

[7] MacNeice, P.: Paramesh: A parallel adaptive mesh refinement community toolkit. Computer physics communications **126** (2000) 330–354

[8] Parashar, M., Browne, J.: System engineering for high performance computing software: The hdda/dagh infrastructure for implementation of parallel structured adaptive mesh refinement. In: IMA Volume on Structured Adaptive Mesh Refinement (SAMR) Grid Methods. (2000) 1–18

[9] Wissink, A.M., Hornung, R.D., Kohn, S.R., Smith, S.S., Elliott, N.: Large scale parallel structured amr calculations using the samrai framework. In: proceedings of SC2001. (2001)

[10] Steensland, J.: Efficient partitioning of structured dynamic grid hierarchies. Doctoral thesis, Scientific computing, Department of Information Technology, University of Uppsala (2002) Uppsala dissertations from the faculty of science and technology 44.

[11] Schloegel, K., Karypis, G., Kumar, V.: A unified algorithm for load-balancing adaptive scientific simulations. In: Proceedings Supercomputing 2000. (2000)

[12] Dreher, J., Grauer, R.: Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws. Parallel Computing **31** (2005) 913–932

[13] Maerten, B.: Drama: A library for parallel dynamic load balancing of finite element applications. In: Lecture Notes in Computer Science 1685. (1999) 313–316

[14] Walshaw, C., Cross, M., Everett, M.: Parallel dynamic graph partitioning for adaptive unstructured meshes. Parallel Distributed Computing **47(2)** (1997) 102–108

[15] Rantakokko, J.: Partitioning strategies for structured multiblock grids. Parallel Computing **26** (2000) 1661–1680

[16] Steensland, J., Söderberg, S., Thuné, M.: A comparison of partitioning schemes for blockwise parallel samr algorithms. In: Lecture Notes in Computer Science 1947. (2001) 160–169

[17] Balsara, D., Norton, C.: Highly parallel structured adaptive mesh refinement using parallel language-based approaches. Parallel Computing **27** (2001) 37–70

[18] Rantakokko, J.: Comparison of parallelization models for structured adaptive mesh refinement. In: Lecture Notes in Computer Science 3149. (2004) 615–623

[19] Ferm, L., Lötsetdt, P.: Space-time adaptive solutions of first order pdes. Journal of Scientific Computing **26**(1) (2006) 83–110

[20] Sun Microsystems `http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf`: Solaris Memory Placement Optimization and Sun Fire servers. (2003)

[21] Teller, P.J.: Tranlation-Lookaside Buffer Consistency. Computer **23**(6) (1990) 26–36

[22] 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

[23] Laudon, J., Lenoski, D.: The SGI Origin: a ccNUMA highly scalable server. In: Proceedings of the 24th annual international symposium on Computer architecture, ACM Press (1997) 241–251

[24] Tikir, M.M., Hollingsworth, J.K.: Using Hardware Counters to Automatically Improve Memory Performance. In: SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2004) 46

[25] Spiegel, A., an Mey, D.: Hybrid Parallelization with Dynamic Thread Balancing on a ccNUMA System. In Brorson, M., ed.: Proceedings of the 6th European Workshop on OpenMP, Royal Institute of Technology (KTH), SWEDEN (2004) 77–81