Int J Parallel Prog (2007) 35:477–491
DOI 10.1007/s10766-007-0056-z

Dynamic Data Migration for Structured AMR Solvers

Markus Nordén · Henrik Löf · Jarmo Rantakokko · Sverker Holmgren

Received: 6 November 2006 / Accepted: 26 January 2007 / Published online: 6 September 2007

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 (AMR). 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.

Keywords Adaptive mesh refinement · SAMR · OpenMP · Geographical locality · cc-NUMA · Page migration · Graph partitioning · Shared memory

M. Nordén · H. Löf · J. Rantakokko · S. Holmgren (✉)
Department of Information Technology, Uppsala University,
Box 337, Uppsalas S-751 05, Sweden
e-mail: sverker@it.uu.se

M. Nordén
e-mail: markusn@it.uu.se

H. Löf
e-mail: henlof@it.uu.se

J. Rantakokko
e-mail: jarmo@it.uu.se

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 large scale shared memory computers are built from nodes with one or several processors, 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 [1–4].

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 communication 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 article, 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?

We also consider an additional question regarding migration, viz., how it should be invoked.

The rest of the article is organized as follows: In Sect. 2 we describe existing parallel SAMR solvers and techniques used for distribution of work and data. In Sect. 3 we consider the model PDE application that is solved, in Sect. 4 we introduce the NUMA computer system used, and in Sect. 5 we give some details about the implementation and experimental setup. In Sect. 6 we present performance results, and in Sect. 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], Chombo [9] and SAMRAI [10] exploit a local address space model implemented using MPI. The parallelization is based on grid patches or blocks, i.e., 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 [11]. 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 [12]. 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., [11]. 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 [6,13,7,8,10], 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 [12,14,15] 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 [16,17]. For the experiments performed in this article, 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 use an algorithm of this type from the Jostle package [15].

In [18], 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 [13] 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 CPUs.

In [19] different programming models using MPI, OpenMP, and hybrid MPI-OpenMP for parallelizing a SAMR solver are compared on a Sun Fire 15000. 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.

Finally, Blikberg et al. [20,21] present two-level nested parallelizations and load balancing algorithms for OpenMP implementations of blockwise AMR. They do not consider data locality and report only modest parallel performance.

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 [22]. However, this would not affect the parallel performance much,

```
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
   Compute next timestep
end do
```

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

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 by a factor two in the entire block. On the contrary, if the error is small enough, the grid in the block is coarsened by 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 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 Fig. 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 later, we perform a total of 20,000 time steps. Adaption, partitioning and migration (if active) is performed every `AdaptInterval` time step, where we use `AdaptInterval`=20. We use a discretization with 16×16 blocks, and the adaption criterion results in three different block sizes: 100×100, 200×200 and 400×400. 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

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

where w_i is the amount of work in partition i and n the number of partitions. Table 1 shows the arithmetic mean of γ_j for the application studied. As we partition every 20th timestep for a total of 20,000 timesteps $j = 1, \ldots, 1{,}000$. As a reference we have also included the results of the MeTiS [23] scratch-remap graph-partitioner. We can see from Table 1 that it becomes very hard to achieve a good load balance with

Table 1 Arithmetical means of the load balance γ_j for $n = 4, 8, 16, 32$ partitions using the Jostle diffusion partitioner and the MeTiS scratch-remap partitioner

n	Diffusion	Scratch-remap
4	1.031	1.026
8	1.059	1.052
16	1.239	1.262

more than eight partitions. This fact might not come as a surprise as we only have 256 blocks which are very heterogeneous in size. The choice of partitioning strategy does not affect the arithmetical load balance very much for this model problem. Because we are interested in minimizing inter-partition data movement, we choose to use the diffusion partitioner.

4 The NUMA System

All experiments presented 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 8 MB 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 code 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 [24]. 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 shoot-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 content 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 CPUs. Due to kernel consistency issues this procedure needs to be serialized using global locking [25,26].

A migrate-on-next-touch directive is also available on the Compaq Alpha Server GS-series [27]. On SGI Origin-systems [28], 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. [29] 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 and an Mey [30] showed how to use the migrate-on-next-touch call to speed up a 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. In [31] we noticed that migration could be invoked in three different ways:

- User *controlled* migration, i.e., that the user specifies exactly where each memory page should reside.
- User *initiated* migration, i.e., that the user specifies that a certain memory page should be migrated but relies on a next-touch mechanism to determine where.
- Transparent migration, i.e., the need for migration is detected and performed automatically, without any intervention from the user.

For adaptive PDE solvers, with a dynamically changing memory access pattern, the situation is more complex than for a PDE solver with a static access pattern.

First of all, for an adaptive PDE solver, with a grid consisting of several smaller blocks, the next-touch strategy may have a problem with "false sharing" of memory pages. If two different threads are responsible for adjacent blocks, the "wrong" thread may be the first to touch a memory page while updating boundary data.

Furthermore, there is no single data distribution that is optimal throughout the entire execution. The grid must be repartitioned dynamically and as a result there may be a need for migration. However, in such situations, a majority of the data may already reside in a favorable part of the physical memory. Thus, the overhead for migration may be reduced by only migrating data that is unfavourably located.

5.1 Experimental Setups

To investigate the performance impact of geographical locality we perform experiments where the application was executed using four threads on the following three 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 transferred to another node. Force immediate migration by touching pages.

In addition, we also compare four different migration strategies for the adaptive PDE solver. They differ in two ways:

Data selectivity (***all*** **or** ***some***), i.e., if *all* shared data is tagged for migration at every adaptation occasion or only *some* memory pages (those belonging to blocks that have been moved to another thread) are tagged for migration.
Affinity strategy (***explicit*** **or** ***on touch***), i.e., if it is stated *explicitly* where a memory page should migrate or if that is decided by the next-touch strategy when the data is accessed some time in the future (on touch).

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 L2-cache misses served by local memory (`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 4,000 times steps of the 20,000 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

6.1 Impact of Geographical Locality

Table 2 shows both total execution time and hardware counter data from the three setups. We can see from Table 2 that the NUMA case runs slower than the UMA case and the NUMA-MIG case. The number of remote accesses is also much higher for

Table 2 Execution time measurements and hardware counter data from the three different experimental setups

	UMA	NUMA	NUMA-MIG
Total execution time	4.09h	6.64h	3.99 h
L2 miss ratio	4.3%	3.9%	4.2%
L2 remote ratio	0.2%	62.9%	8.1%

The L2 Remote ratio is the ratio of cache misses served by remote memory to the total number of L2 misses

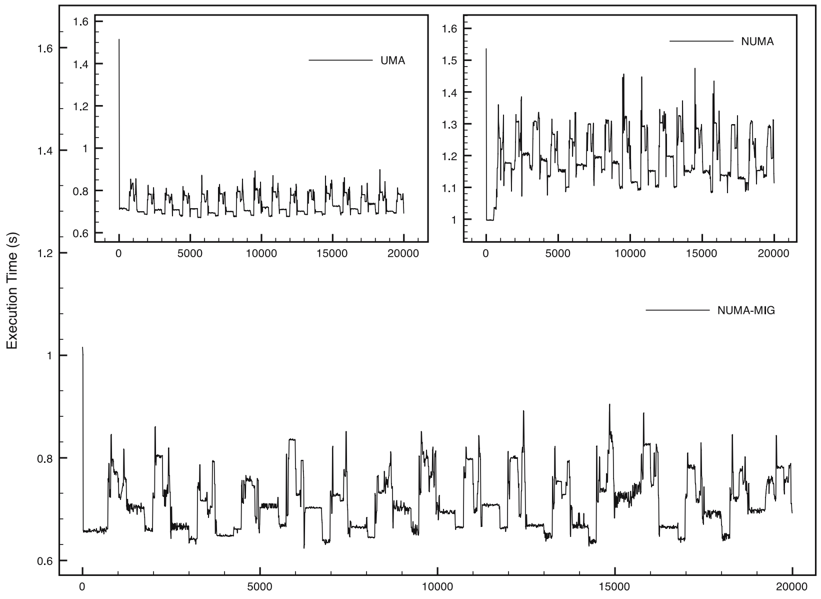
this case compared to the UMA and NUMA-MIG cases which shows that the NUMA case exhibits a low degree of geographical locality. It is also clear that the effect of page migration is large since the amount of remote accesses for the NUMA-MIG case is much lower compared to the NUMA case. We cannot completely 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. In Fig. 2a the execution time for the UMA and NUMA cases are displayed on top of the graph for the NUMA-MIG case. In Fig. 2b only the first 2,000 time steps are shown and the three graphs are aligned in time. 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 CPUs 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.

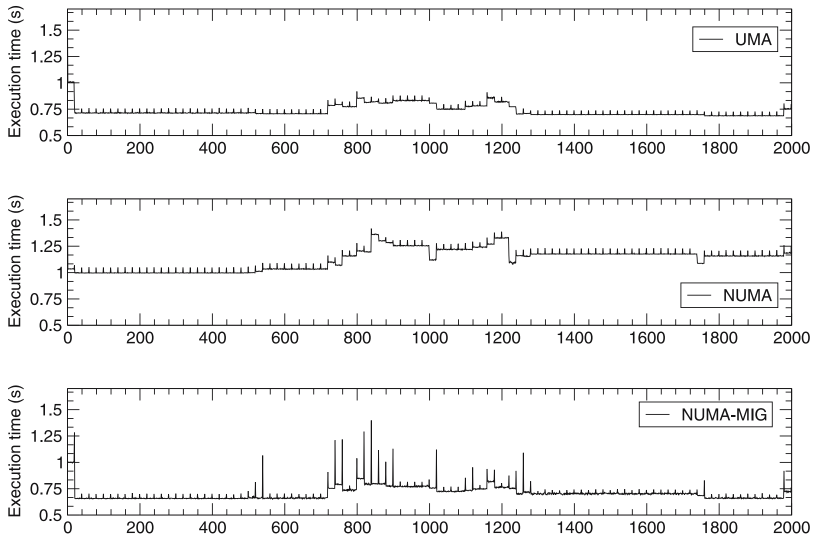
Table 3 shows migration statistics for the application in the NUMA-MIG case. At a typical iteration the resident working set is about 350 MB which corresponds to 44800 8kB pages. Migration is triggered every 20th time step which gives a total of 1,000 times. Assuming that the migrations, 2,212,844 pages in total, are evenly distributed over time, only 2,213 pages (18.1 MB) are migrated at each migration. This small amount of pages correspond to about 5% of a typical working set. Hence, we conclude that the amount of data migrated at each migration is fairly low. This together with the fact that the NUMA-MIG case executes faster than the UMA case indicates that the overhead of migration is 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 min) in the Solaris page fault trap handler. This fact further supports the conclusion that the overhead from migration is low.

6.2 Different Migration Strategies

Results, in terms of execution times for the four different migration strategies, are found in Table 4.



(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 20th time step

Table 3 Number of 8kB pages migrated for the NUMA-MIG case collected using Solaris kernel statistics (kstat)

	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)

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 i.e., the total amount of migrations for one node. The total amount of migrated pages for all nodes was 2212844 (16.88 GB)

Table 4 Execution time for different migration strategies

Data selectivity	Affinity strategy	Execution time (h)
Some	Explicit	3.99
All	Explicit	4.37
Some	On touch	4.49
All	On touch	4.51

The two migration strategies with explicit affinity are examples of user controlled migration, as defined in Sect. 5, and the two with affinity on touch are user initiated. Since there is no support for transparent migration on the SF15k system, we were not able to study any such migration strategy.

The results show that the execution time is reduced significantly for all four migration strategies in comparison with the NUMA case which took 6.64 h, see Table 2. The main conclusion is thus that the decision to migrate at all is more important than which migration strategy to use.

There is, however, a difference in execution time also between the different migration strategies. When migrating all data, the overhead for migration is higher than when migrating only misplaced data and when using the on touch affinity, the data distribution will in general not become optimal due to stealing of pages belonging to the ghost nodes.

From a programmer perspective, user initiated migration—i.e., using on touch affinity—of all data is most appealing since it is the least demanding to implement, but it also gives the least improvement of execution time.

Improving only the data selectivity, i.e., migrating misplaced data with on touch affinity, is a somewhat contradictory strategy. On one hand, it does not require that the programmer specifies where to migrate a memory page but on the other hand, he is assumed to know which memory pages are misplaced. This strategy is only marginally better than migrating all data with on touch affinity.

Migrating all data explicitly can be seen as a compromise, where the programmer is relieved of the burden of knowing anything about the previous data distribution but where he has to specify the new data distribution. In this case, the data distribution will be optimal but there will be unnecessary overhead from trying to migrate memory pages that are already favorably placed.

Finally, fully user controlled migration, i.e., migrating only misplaced data with explicit affinity specifications, gives an optimal data distribution with minimal migration overhead. As expected, this migration strategy gives the best improvement of execution time, but it is not indisputable that the improvement is worth all the extra implementation work compared to user initiated migration of all data.

6.3 Multiple Threads Per Node

All of the experiments presented so far have been executed using four threads on four nodes to be able to compare with the corresponding UMA case. Using four threads we found that we could increase the amount of geographical locality and that the overhead of page migration was low. As the primary reason for implementing cc-NUMA architectures is to increase the scalability of the system we would like to study the geographical locality when more than four threads are used.

In the previous experiments partitioning for the threads is the same as partitioning for the nodes. If we increase the number of threads per node we have the choice of partitioning for the threads and mapping several partitions to each node or partitioning for the nodes and then assign multiple threads to each partition. The latter strategy requires a hierarchical parallelization which we have not yet implemented. Instead we choose to study the multiple-partitions per node case.

We executed the code using the diffusion partitioner, “some” data selectivity option and an explicit data affinity strategy for 2, 3, and 4 threads per node. The partitions were mapped to the four nodes in a linear fashion. We can see from Table 5 that we can improve the degree of geographical locality using page migration in the multiple threads per node case. We also see that the scalability is rather poor. There are several potential explanations for this behavior. First, the load balance of the application is not very good when we increase the number of threads. For 16 threads the load imbalance is 24%, see Table 1. The AMR strategy used for the experiments use a fixed number of grid blocks. Increasing this number can be a way of improving the load balance. However, the number of blocks are constrained by factors determining AMR-efficiency such the ability to capture regions of the domain needing resolution. These factors are often more important than achieving a good load balance for typical AMR problems. The low number of pages migrated in the 16 thread case can be explained by the fact that the workload is heterogenous. When a total of 256 blocks are used and the heaviest blocks have the weight 16, some of the partitions contained only two heavy blocks. In some cases, the neighboring blocks very also heavy resulting in that the diffusion algorithm did not repartition.

Second, the available bandwidth to the memory system is consumed as we increase the number of threads. The UMA experiments performed earlier indicated that a single

Table 5 Execution times in hours using multiple threads per node

Threads per node	NUMA	NUMA-MIG	Pages migrated
1	6.64	3.96	2,212,844
2	3.51	2.43	2,870,340
3	3.04	1.87	3,099,778
4	2.35	1.67	908,274

The total amount of threads is 4, 8, 12, and 16. The Pages migrated column shows the total number of pages migrated to and from a node during the whole simulation

node could not tolerate the bandwidth demand of four threads. Furthermore, when all CPUs are used other system processes interfere with the execution.

Finally, the overhead of page migration might increase as we increase the number of threads. In a previous study, Löf and Holmgren [26] showed that the overhead of page migration increased with the number of threads. However, in that study the migration was triggered once and the entire dataset of about 62,000 8kB pages were migrated. In this application the total amount of migrated pages is distributed over the entire execution time. Following the discussion in Sect. 6.1, the amount of pages migrated at a single invokation is our case much lower, about 2–3000 pages. However, overhead from migration is not only determined by the number of pages migrated but also by the number of TLBs needed to keep consistent, see Sect. 4. As the number of threads increase it is likely that the overhead from page migration also increase.

Hence, we cannot rule out the probability that overhead from migration affects the execution time in a negative way. However, we believe that the main reason for the poor scalability is the poor load balance.

7 Conclusions

In this article 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 overhead of migration was found to be low which can be attributed to two facts. First, our experiments were performed using a moderate amount of threads and the datasets were not very large. The overhead from page migration will probably increase with the size of the dataset and the number of nodes and CPUs. Second, for SAMR to be economic 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 fairly 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 and dynamics 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 need 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.

References

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, pp. 33–33. ACM Press, New York, NY, USA (2001)
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, pp. 121–129. ACM Press (2003)
3. Holmgren, S., Nordén, M., Rantakokko, J., Wallin, D.: Performance of PDE solvers on a self-optimizing NUMA architecture. Parallel Algor. Appl. **17**(4), 285–299 (2002)
4. Mark Bull, J., 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, hierarchical adaptive mesh refinement algorithms. Comput Visual Sci **3**, 147–157 (2000)
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, pp. 361–372. Springer (2003)
7. MacNeice, P.: Paramesh: a parallel adaptive mesh refinement community toolkit. Comput Phys Communi **126**, 330–354 (2000)
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, pp. 1–18 (2000)
9. Colella, P., Graves, D.T., Ligocki, T.J., Martin, D.F., Modiano, D., Serafini, D.B., Straalen, B.V.: Chombo Software Package for AMR Applications – Design Document. Applied Numerical Algorithms Group, NERSC Division, Lawrence Berkeley National Laboratories (2000)
10. 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)
11. Steensland, J.: Efficient partitioning of structured dynamic grid hierarchies. Doctoral thesis. Scientific Computing, Department of Information Technology, University of Uppsala. Uppsala dissertations from the Faculty of Science and Technology 44 (2002)
12. Schloegel, K., Karypis, G., Kumar, V.: A unified algorithm for load-balancing adaptive scientific simulations. In: Proceedings Supercomputing 2000 (2000)
13. Dreher, J., Grauer, R.: Racoon: a parallel mesh-adaptive framework for hyperbolic conservation laws. Parallel Comput. **31**, 913–932 (2005)
14. Maerten, B.: Drama: a library for parallel dynamic load balancing of finite element applications. In: Lecture Notes in Computer Science, Vol. 1685, pp. 313–316 (1999)
15. Walshaw, C., Cross, M., Everett, M.G.: Parallel dynamic graph partitioning for adaptive unstructured meshes. Parallel Distributed Comput. **47**(2), 102–108 (1997)
16. Rantakokko, J.: Partitioning strategies for structured multiblock grids. Parallel Comput. **26**, 1661–1680 (2000)
17. Steensland, J., Söderberg, S., Thuné, M.: A comparison of partitioning schemes for blockwise parallel samr algorithms. In: Lecture Notes in Computer Science, Vol. 1947, pp. 160–169 (2001)

18. Balsara, D.S., Norton, C.D.: Highly parallel structured adaptive mesh refinement using parallel language-based approaches. Parallel Comput. **27**, 37–70 (2001)
19. Rantakokko, J.: Comparison of parallelization models for structured adaptive mesh refinement. In: Lecture Notes in Computer Science, Vol. 3149, pp. 615–623 (2004)
20. Blikberg, R.: Nested Parallelism in OpenMP with Application to Adaptive Mesh Refinement. PhD thesis, Parallab/Department of Informatics, University of Bergen, Norway, Februariy 2003 (2003)
21. Blikberg, R., Sørevik, T.: Load balancing and openmp implementation of nested parallelism. Parallel Comput. **31**(10-12), 984–998 (2005)
22. Ferm, L., Lötsetdt, P.: Space–time adaptive solutions of first order pdes. J. Sci. Comput. **26**(1), 83–110 (2006)
23. Karypsis, G., Kumar, V.: A fast and highly qualitymultilevel scheme for partitioning irregular gra phs. SIAM J. Sci. Comput. **20**(1), 359–392 (1999)
24. Sun Microsystems, http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf. Solaris Memory Placement Optimization and Sun Fire servers, January 2003 (2003)
25. Teller, P.J.: Translation-lookaside buffer consistency. Computer **23**(6), 26–36 (1990)
26. Löf, H., Holmgren, S.: Affinity-on-next-touch: increasing the performance of an industrial pde solver on a cc-numa system. In: ICS ’05: Proceedings of the 19th Annual International Conference on Supercomputing, pp. 387–392. ACM Press, New York, NY, USA (2005)
27. Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Alexander Nelson, C., Offner, C.D.: Extending OpenMP for NUMA machines. Sci. Program, **8**, 163–181 (2000)
28. Laudon, J., Lenoski, D.: The SGI Origin: a ccNUMA highly scalable server. In: Proceedings of the 24th Annual International Symposium on Computer architecture, pp. 241–251. ACM Press (1997)
29. 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, p. 46. IEEE Computer Society, Washington, DC, USA (2004)
30. 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, pp. 77–81. Royal Institute of Technology (KTH), Sweden (2004)
31. Löf, H., Nordén, M., Holmgren, S.: Improving geographical locality of data for shared memory implementations of PDE solvers. In: Sloth, P.M.A., Tan, C.J.K., Dongarra, J.J., Hoekstra, A.G. (eds.) Computational Science – ICCS 2004, Part II, pp. 9–16. Springer-Verlag, Berlin (2004)