

affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System

Henrik Löf and Sverker Holmgren
Uppsala University
Department of Information Technology
Box 337, 752 37 Uppsala, SWEDEN
{henrik.lof,sverker.holmgren}@it.uu.se

Abstract

To achieve close to optimal performance on cc-NUMA systems for shared memory parallel applications with complex data access patterns, a mechanism for co-locating threads and the data during the execution of the program is needed. The affinity-on-next-touch procedure studied in this paper is based on re-doing the standard first-touch allocation at explicitly given locations in the code. We study the performance of a parallelized scientific computing application for which thread-data affinity can not be created by standard methods. We observe a performance improvement of 64% if the affinity-on-next-touch procedure is invoked once at a critical location in the code. We also perform experiments that show that the overhead connected to creating the affinity can be almost fully attributed the handling of page entries in the TLBs. The cost for actually migrating data is negligible. Using larger but fewer pages we measure a performance improvement of 127% compared to the original code.

1. Introduction

The memory access patterns of scientific computing codes are becoming more and more complex. Unstructured grids and adaptive schemes are introduced in the computational models to increase the accuracy and resolution, resulting in that the data access patterns are more irregular and possibly also changes during execution. When parallelizing such codes, a shared memory programming model is preferred. One reason is that the programmer does not have to explicitly manage the communication between processors. Another important reason is that programming tools like OpenMP use a uniform shared memory model, implying that the programmer does not have to specify how the data should

be partitioned and where it should be stored in the memory. However, modern scalable shared memory computers are normally of cc-NUMA type. In such systems, memory and CPUs are distributed over several nodes, where each node normally consists of a physical CPU-memory board. The access time for memory within a node is smaller than the time required for a remote memory access, and the difference is measured by the NUMA-ratio,

$$\text{NUMA-ratio} = \frac{\text{Remote access time}}{\text{Local access time}}.$$

On a NUMA system, the distribution of data in the shared memory has an effect on the performance of programs. Misplacement of data may cause that the performance and/or scalability is reduced even if the code is highly optimized in other aspects.

Possible ways of reducing negative performance effects arising from the mismatch between the programming model and the underlying computer architecture has been discussed by many authors, see e.g. [2, 13, 5, 9, 12]. The proposals fall into two main categories:

Extend the programming model: Here, the programmer may explicitly specify the data distribution using directives that are added to the programming model, see e.g [1]

Extend the computer system: Here, the OpenMP runtime system, the operating system and/or special hardware provide mechanisms for detecting and correcting misplacement of data in a transparent way while the programming model is left unmodified [16, 14, 13, 3].

Both of these approaches have advantages and disadvantages. If the programmer knows how data is accessed, it may be easy to explicitly specify a good data distribution. Then, a programming model where this distribution can be described directly will probably result in good performance. On the other hand, if the data access pattern is complex, a

substantial effort may be needed by the programmer to determine a good data distribution, or it may not even be possible. In such situations, a solution where the computer system adaptively determines the data distribution in a transparent way may be more beneficial. However, there is a risk that the overhead introduced by the modifications of the system will result in a loss of performance and/or scalability, counteracting positive effects of improving the data distribution.

In this paper, we study a method for dealing with the data distribution problem based on a run-time mechanism for co-locating data and the threads that use it on the same node of the computer system. The mechanism can be made available to the programmer by a portable and simple addition to the programming model. We evaluate the performance effect of applying this method to an industrial scientific application, parallelized using OpenMP and executed on a cc-NUMA computer. The results show that it is possible to decrease the execution time significantly compared to if the original code is used. The results also show that, on the computer system studied, the overhead introduced by the directive is dominated by handling TLB coherence and not by the actual movement of data. This implies that the overhead can be reduced by using large page sizes.

2. Thread-data affinity on cc-NUMA systems

When executing a parallel program on a cc-NUMA system, it must be determined how the data should be partitioned over the nodes and at which nodes the threads should be run. To get high performance, the number of accesses to remote memory should be small. This can be achieved if the *thread-data affinity* is large, i.e. if each data item resides in the memory of the node where the thread that uses it the most is running.

The standard method for creating thread-data affinity is to use a first-touch strategy. When a memory page is accessed for the first time during execution, the operating system has to reserve a physical page for it in one of the nodes. Using first-touch data placement, the page is allocated in the node where the thread responsible for the first access is running. The hope is that this thread will also be the one that accesses the page the most later, and that the thread will continue to run in the node where it was started.

If the data access pattern changes after the initialization or if the threads move between nodes during execution, the thread-data affinity is normally reduced. The first situation occurs for example if the data is initialized by a single thread before the parallel region in the program begins. In such cases the first-touch strategy normally results in that all data is allocated on a single node. This problem might sometimes be tackled by rewriting the code so that it performs a parallel initialization of the data, see e.g. [10, 9].

However, in other cases a substantial programming effort is needed to achieve this. Furthermore, some form of indirect addressing is used in many codes, for example for handling unstructured computational grids. Then, the data access pattern is determined by the data itself, and it is not possible to exploit the standard first-touch strategy in an optimal way. In such cases, a viable approach for creating thread-data affinity is to introduce a mechanism for re-doing the first-touch placement of data at some locations in the code during the execution. This would allow a re-creation of thread-data affinity by effectively migrating data to the correct nodes. We use the term *affinity-on-next-touch* for such an operation.

If the access pattern of the main part of the program is static, it might be sufficient to invoke the affinity-on-next-touch procedure once after the initialization. For example, for an application using an unstructured grid, the procedure could be invoked for migrating data to the correct nodes once the access pattern is determined. In general scientific computing codes, a few distinct phases with different access patterns can often be identified. It is normally easy for the programmer to determine where affinity should be created, which implies that it is reasonable to add an affinity-on-next-touch directive to the programming model. A directive of this type is also mentioned in [14], and a first implementation is found in the Compaq OpenMP compilers [1]. An affinity-on-next-touch directive does not specify how and where to migrate data, which makes it portable and easy to use. Moreover, the time-consuming phases in computational codes normally consist of iterations where the data structures are repeatedly accessed using the same access pattern. In such cases, the overhead introduced by invoking the directive before the loop is amortized over the iterations.

If the data access pattern changes more often during the execution, the affinity-on-next-touch mechanism could be invoked repeatedly, or a fully transparent scheme for adaptive data distribution could be used. In a transparent scheme, ways of detecting candidate pages for migration and/or replication must be built into the system, and algorithms for deciding where and when to move data must be frequently executed. When implementing schemes of this type, great care must be taken not to introduce significant amounts of overhead.

3. The application

To evaluate the applicability of an affinity-on-next-touch directive, we study the performance of an industrial solver for the Maxwell equations in 3 dimensions, describing the scattering of electromagnetic waves from e.g. an airplane [7]. The solver uses an unstructured grid and a finite element discretization, and solves the resulting sparse system

of equations using a conjugate gradient (CG) solver. In the iterative solver, a bandwidth minimization algorithm [8] has been introduced [11]. The code is written in Fortran 90 and C, and parallelized using OpenMP.

The data access pattern in the computations is static and unstructured, and it is possible identified two main phases in the code:

Assembly of iteration matrix This part of the code is not very time-consuming. Also, it is rather complex and hence not parallelized

Iterative solver This computationally heavy part of the code is parallelized using an algorithm with a small number of synchronization points [11]. For normal application problems, about 50 iterations are required to compute the solution

In the experiments presented below we study a problem where the iteration matrix has 1794058 rows, and each row has on average 15 non-zeros entries. This leads to that the size of the data set is approximately 500 MByte.

Since the assembly of the iteration matrix is performed by a single thread, all of the data will be allocated to a single node after the assembly phase. By invoking the affinity-on-next-touch procedure just before the first iteration in the iterative solver, the data will be migrated to several nodes matching the location of the multiple threads used in the parallel region. In our experiments, we run the iterative solver to convergence once, and investigate the performance gain from creating affinity for this computation. In a real-life computation, the system of equations is solved over and over again for hundreds of time steps, implying that the overhead associated with the creation of thread-data affinity will be amortized over many more iterations.

4. The cc-NUMA system

The experiments are performed on a dedicated 32 CPU domain on a Sun Fire 15000 system (SF15K) [4]. Here, each node consists of four 900MHz UltraSPARC-III CPUs, each having 8 MByte of L2 cache and 4 GByte of main memory. The system runs Solaris 9, where the affinity-on-next-touch mechanism can be implemented by a call to the `madvise(3C)` library routine.

In the experiments, the setting of a kernel parameter for the Solaris scheduler is modified to increase the probability that the threads stay on the same nodes during the execution of the program [15]. We did not use any binding to CPUs or processor sets.

It should be noted that NUMA-ratio of the SF15K system is only approximately two. This means that we can not expect that the effects of modifying the distribution of data to be dramatic.

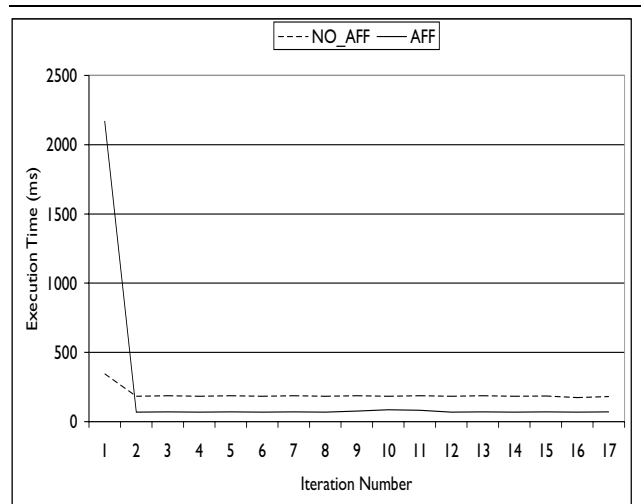


Figure 1. The effect of the affinity-on-next-touch procedure when 16 threads are used. The directive was executed just before the first iteration. Only the first 17 out of 47 iterations are shown.

5. Results

In Figure 1, the effect of invoking the affinity-on-next-touch mechanism is illustrated. In this experiment, 16 threads are used. The graph shows the execution time for each iteration if the original code is used (`NO_AFF`) and if the affinity-on-next-touch mechanism is introduced before the first iteration in the solver (`AFF`).

In the `NO_AFF` case, the data remains allocated on a single node. The number of remote accesses is large, resulting in a rather large execution time per iteration and a total execution time of 9.35 s. The slightly larger execution time for the first iteration can be attributed to cold-start effects, where some of the data is cached on multiple processors when the parallel region is entered.

For the `AFF` case, it is clear that an affinity-on-next-touch directive affects the performance in two ways. In the first iteration, there is a cost for creating the affinity and the execution time for this iteration is increased. After this, the execution time per iteration is significantly reduced. Using the UltraSparc-III hardware counters, a measurement of the number of L2 cache misses that are served from remote memory verifies that the data is actually migrated to the nodes involved in the parallel computations. Therefore, we conclude that the thread-data affinity actually has a positive effect on performance. The average execution time for iteration 2 – 47 is reduced from 185 to 71 ms, and the total execution time is reduced to 5.91 s.

In a more general situation, it is clear that the initial cost

of creating affinity must be amortized over a number of iterations (or similar) if the total performance should be increased. A simple calculation shows that for our experiment, 17 iterations are needed before the break-even point is reached.

It is interesting to study of the overhead for the affinity-on-next-touch procedure in more detail. In the experiment presented in Figure 1, we know that data is actually migrated from one node to several others at the first iteration. The overhead for this can be divided into two parts: The cost of copying data and the cost of keeping the page tables coherent. The copy operations are parallelizable, and the performance is only limited by the memory system characteristics. However, the modifications of the page tables are potentially more problematic.

If a page translation is cached in several translation look-aside buffers (TLBs), we need to enforce coherency among these buffers. This procedure is serial, and involves an expensive *TLB shoot-down* process where dirty translations in the TLBs of processors sharing the page are replaced. This process often involves global kernel locks, polling and/or interrupts [6]. To investigate the relative effect of the two types of overhead, another experiment was performed where the affinity-on-next-touch procedure is invoked twice, once at iteration 15 and once at iteration 40. Since the access pattern is static during the iterations, no data is actually moved on the second invocation. The result show that there was no measurable time-difference iterations 15 and 40, and it is clear that the overhead is dominated by handling the TLBs. This assumption is further supported by results from measuring the number of minor page faults and cross-processor interrupts using kernel statistics.

As described above, we have found that the overhead connected to creating thread-data affinity is dominated by handling entries in the TLBs. It is reasonable to assume that using fewer but larger pages could reduce this overhead. In Figure 2, the effect of using 64kB pages instead of the standard 8kB pages is show, again for an experiment using 16 threads. In the figure, the two curves in Figure 1 are repeated (NO_AFF8kB and AFF8kB), but now together with the corresponding curves for 64kB pages (NO_AFF64kB and AFF64kB). It is clear that the overhead in the first iteration is reduced by a factor of 3 when using 64kB pages compared to the 8kB case. Also, the performance for the subsequent iterations is almost unaffected by changing the page size. The total execution time is now reduced to 3.9 s, and the break-even point where the overhead has been amortized is reached already after four iterations. The primary reason for the reduction of the overhead when using a larger page size is that fewer TLB entries must be handled. In the TLB shoot-down process, the initiating thread sends one interrupt per page to invalidate possible shared TLB entries for other threads. The UltraSparc-III processor

does not have hardware support for 64kB pages, implying that when 64 kB pages are used, in practice one interrupt invalidates eight 8kB pages. The results for the original code in Figure 2 indicate that there is another, smaller positive effect provided by a general reduction of the cold-start effects for entering the parallel region when the number of TLB entries needed to cache the working set is reduced.

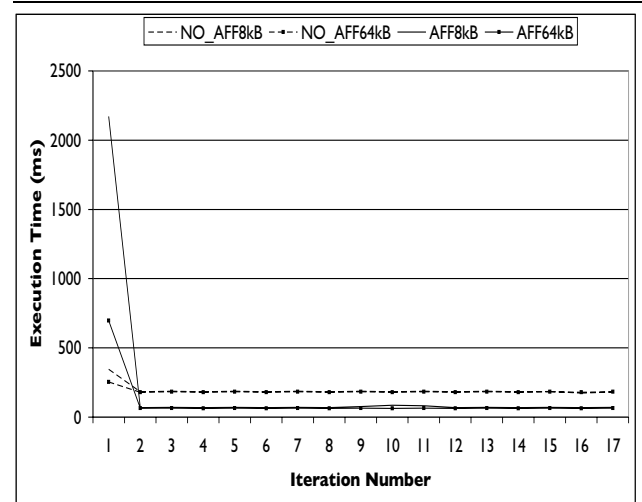


Figure 2. A comparison of the effect of the affinity-on-next-touch procedure when 8kB and 64kB page are used.

In Figure 3, the effect of invoking the affinity-on-next-touch procedure for different numbers of threads is shown. The bars show the improvement of the total execution time when using 8kB and 64kB pages compared to the cases where the original code is executed using 8kB pages using the same number of threads. From the figure, it is clear that enforcing a proper distribution of data by invoking an affinity-on-next-touch procedure in general improves the performance for the application studied here. Also, it is clear that in all cases the performance improvement is larger when 64kB pages are used instead of 8kB pages. The performance decrease for 2 threads and 8kB pages is explained by that in this case the two threads are scheduled to the same node, and there will be no possible benefit of migrating data. Then, the overhead in the first iteration is not followed by any performance improvement in subsequent iterations.

A more detailed study of the experiments described by Figure 3 reveals that, using the implementation studied in this paper, the affinity-on-next-touch mechanism is not scalable. The initial iteration overhead is almost constant as the number of threads is increased. As the iterations become faster when more threads are used, more and more iterations must be performed before the break-even point is reached

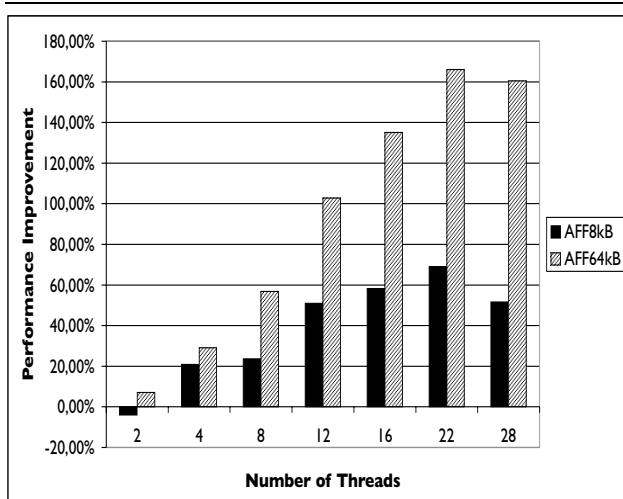


Figure 3. Performance improvement of using the affinity-on-next-touch procedure compared to if the original code is used. Results for both 8kB and 64kB pages are shown.

and the initial overhead is amortized. However, this effect is reduced by using 64kB pages.

Finally, we present a traditional speedup graph where the code is executed using different numbers of threads and the affinity-on-next-touch procedure is invoked as described earlier. Results for both 8kB and 64kB pages are shown. 4. From the figure, it is obvious that reducing the initial iteration overhead has a positive effect on the scalability of the parallel implementation.

6. Conclusions

To achieve close to optimal performance on cc-NUMA systems for shared memory applications with complex and possibly changing data access patterns, a mechanism for creating or re-creating thread-data affinity during the execution of the program is needed. In this paper, we have evaluated such a procedure, denoted *affinity-on-next-touch*, based on re-doing the standard first-touch allocation at given locations in the code. This normally results in that data is migrated between the nodes in the system. The experiments are performed using a parallelized scientific computing application where thread-data affinity can not be created by standard methods. We find that the execution time can be significantly reduced by invoking the affinity-on-next-touch procedure. We also perform experiments that show that the overhead connected to creating the affinity can be almost fully attributed to the handling of page entries in the TLBs. The cost for actually migrating data is negligible. Led by this observation, we manage to reduce the overhead by in-

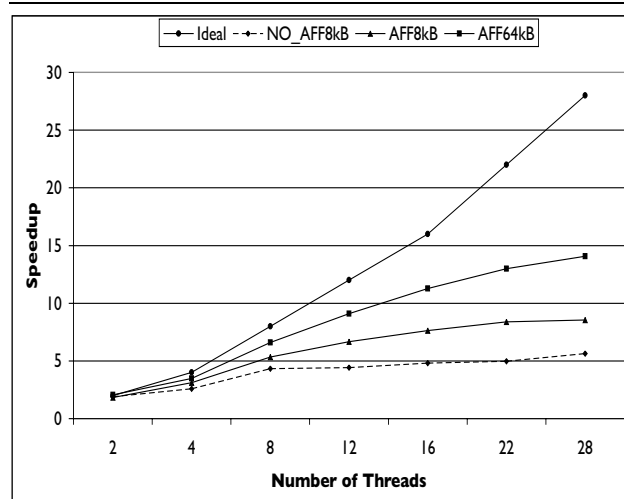


Figure 4. The speedup of the solver using the original code (NO_AFF8kB), affinity-on-next-touch and 8k pages (AFF8kB), and affinity-on-next-touch and 64k pages (AFF64kB).

creasing the page size, which results in that fewer TLB entries must be handled.

Since the affinity-on-next-touch procedure is invoked at distinct locations in the algorithm, it is natural to suggest that it should be accessible by a directive in programming tools like OpenMP. The programmer usually knows where thread-data affinity is critical for performance. In OpenMP, other options are to add an affinity-creation attribute to the parallel region directive, or to automatically invoke the procedure for creating affinity at the beginning of each parallel region. The latter solution has the advantage that the programming model is left unmodified.

References

- [1] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner. Extending OpenMP for NUMA machines. *Scientific Programming*, 8:163–181, 2000.
- [2] T. Brecht. On the Importance of Parallel Application Placement in NUMA Multiprocessors. In *Proceedings of the Fourth Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, San Diego, CA, Sept 1993.
- [3] J. M. Bull and C. Johnson. Data Distribution, Migration and Replication on a cc-NUMA Architecture. In *Proceedings of the Fourth European Workshop on OpenMP*. <http://www.caspar.it/ewomp2002/>, 2002.
- [4] A. Charlesworth. The sun fireplane system interconnect. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 7–7. ACM Press, 2001.

- [5] J. Corbalan, X. Martorell, and J. Labarta. Evaluation of the memory page migration influence in the system performance: the case of the SGI O2000. In *Proceedings of the 17th annual international conference on Supercomputing*, pages 121–129. ACM Press, 2003.
- [6] D. Culler, J. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufman, 1998.
- [7] F. Edelvik. *Hybrid Solvers for the Maxwell Equations in Time-Domain*. Doctoral thesis, Mathematics and Computer Science, Department of Information Technology, University of Uppsala, may 2002. <http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-2156>.
- [8] N. E. Gibbs, J. William G. Poole, and P. K. Stockmeyer. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2):236–250, April 1976.
- [9] S. Holmgren, M. Nordén, J. Rantakokko, and D. Wallin. Performance of PDE Solvers on a Self-Optimizing NUMA Architecture. *Parallel Algorithms and Applications*, 17(4):285–299, 2002.
- [10] H. Jin, M. Frumkin, and J. Yan. The OpenMP Implementation of NAS Parallel Benchmarks and Its Performance. NAS Technical Report NAS-99-011, NASA Ames Research Center, 1999.
- [11] H. Löf and J. Rantakokko. Algorithmic Optimizations of a Parallel Industrial CEM Solver. Technical report, Dept. of Information Technology, Uppsala University, 2004.
- [12] T. G. Mattson. How good is OpenMP. *Scientific Programming*, 11:81–93, 2003.
- [13] D. S. Nikolopoulos, T. S. Papatheodorou, C. D. Polychronopoulos, J. Labarta, and E. Ayguade. A transparent runtime data distribution engine for OpenMP. *Scientific Programming*, 8:143–162, 2000.
- [14] L. Noordergraaf and R. van der Pas. Performance experiences on Sun’s Wildfire prototype. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*, page 38. ACM Press, 1999.
- [15] Sun Microsystems, http://www.sun.com/servers/wp/docs/mpo_v7_CUSTOMER.pdf. *Solaris Memory Placement Optimization and Sun Fire servers*, January 2003.
- [16] B. Verghese, S. Devine, A. Gupta, and M. Rosenblum. Operating system support for improving data locality on CC-NUMA compute servers. In *Proceedings of the seventh international conference on Architectural support for programming languages and operating systems*, pages 279–289. ACM Press, 1996.