Factors Impacting Performance of Multithreaded Sparse Triangular Solve

Michael M. Wolf and Michael A. Heroux and Erik G. Boman

Scalable Algorithms Dept., Sandia National Laboratories, Albuquerque, NM, USA.
{mmwolf, maherou, egboman}@sandia.gov

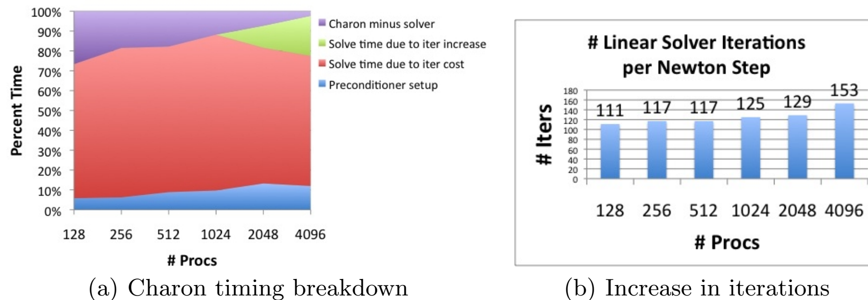
Abstract. As computational science applications grow more parallel with multi-core supercomputers having hundreds of thousands of computational cores, it will become increasingly difficult for solvers to scale. Our approach is to use hybrid MPI/threaded numerical algorithms to solve these systems in order to reduce the number of MPI tasks and increase the parallel efficiency of the algorithm. However, we need efficient threaded numerical kernels to run on the multi-core nodes in order to achieve good parallel efficiency. In this paper, we focus on improving the performance of a multithreaded triangular solver, an important kernel for preconditioning. We analyze three factors that affect the parallel performance of this threaded kernel and obtain good scalability on the multi-core nodes for a range of matrix sizes.

1 Introduction

1.1 Motivation

With the emergence of multi-core processors, most supercomputers are now hybrid systems in that they have shared memory multi-core nodes that are connected together into a larger distributed memory system. Although for many numerical algorithms the traditional message programming model is sufficient to obtain good scalability, some numerical methods can benefit from an hybrid programming model that uses message passing between the nodes with a shared memory approach (e.g., threads) within the node. Scalable threaded algorithms that run efficiently on the node are essential to such a hybrid programming model and are the emphasis of our work in this paper.

Solvers are a good example of numerical algorithms that we believe can benefit from a hybrid approach. Solver implementations based on a flat MPI programming model (where subcommunicators are not utilized) often suffer from poor scalability for large numbers of tasks. One difficulty with these approaches is that with domain decomposition based preconditioners, the number of iterations per linear solve step increase significantly as the number of MPI tasks (and thus the number of subdomains) becomes particularly large. We also see this behavior with scalable preconditioners such as an algebraic multilevel preconditioner. Figure 1 shows an example of this difficulty for Charon, a semiconductor device simulation code [1–3], with a three level multigrid preconditioner.



(a) Charon timing breakdown (b) Increase in iterations

Fig. 1. Strong scaling analysis of Charon on Sandia Tri-Lab Linux Capacity Cluster for 28 million unknowns.

As the number of MPI tasks increases, the number of linear solver iterations increases (Figure 1(b)). Figure 1(a) shows that these extra iterations require an increasingly higher percentage of the total runtime as the number of MPI tasks increase, resulting in a degradation in the parallel performance.

By having fewer (but larger) subdomains, better convergence can be obtained for the linear solver. With fewer subdomains, the solvers for these larger subdomains must be parallel in order to maintain the overall scalability of the algorithm. This leads to a two-level model of parallelism, where MPI is used to communicate between subdomains and a second level of parallelism is used within each subdomain. One approach is to also use MPI to obtain parallelism at the subdomain level (e.g., [4]). Another approach, which we explore in this paper, utilizes multithreading to obtain parallelism at the subdomain level. This approach is limited in that each subdomain does not extend beyond the processor boundaries. However, we feel that as the number of cores per processor continues to increase, this will become less important and threads may be a better approach for exploiting the shared memory architecture on the node.

Keeping the iteration count low is not sufficient, however, to obtain performance gains over MPI-only implementations. The shared memory numerical kernels that run on each multi-core node also need to be scalable. It is particularly important to have a scalable shared memory implementation of a triangular solver to run on each node since this kernel will be executed for each iteration of the linear solver. The focus of this paper is to study the various factors that affect the performance of this shared memory triangular solver kernel in the pursuit of a sufficiently scalable algorithm.

1.2 Level-set triangular solver

We focus our attention on improving the performance of a level-set triangular solver for sparse matrices as described in [5]. Below we describe the process for lower triangular matrices, but the upper triangular case is analogous. First, we express the data dependencies of the triangular solve for the lower triangular

matrix $\mathbf{L}$ as a directed acyclic graph (DAG). A vertex v_i of this DAG correspond to the vector entry x_i that will be calculated in the triangular solve. The directed edges in the DAG represent data dependencies between the $\mathbf{x}$ vector entries, with a directed edge connecting v_i to v_j if and only if x_i is needed to compute x_j. The level-sets of this DAG represent sets of row operations in the triangular solve operation that can be performed independently. Specifically, the ith level-set is a set of vertices that have incoming edges only from vertices of the previous $i-1$ levels (the corresponding x_i entries are only dependent on x vector entries in previous levels). We calculate these level-sets from the DAG using a variant of breadth-first search. Permuting the system symmetrically so that the rows/columns are in order of the level-sets, we obtain the following permuted matrix,

$$\tilde{\mathbf{L}} = \mathbf{PLP}^T = \begin{bmatrix} \mathbf{D}_1 & & & & \\ \mathbf{A}_{2,1} & \mathbf{D}_2 & & & \\ \mathbf{A}_{3,1} & \mathbf{A}_{3,2} & \mathbf{D}_3 & & \\ \vdots & \vdots & \vdots & \ddots & \\ \mathbf{A}_{l,1} & \mathbf{A}_{l,2} & \mathbf{A}_{l,3} & \dots & \mathbf{D}_l \end{bmatrix},$$

where l is the number of level-sets. This symmetrically permuted matrix is still triangular since $\tilde{x}_i$ can only depend on those $\tilde{x}_j$ calculated in a previous level with this dependency corresponding to a lower triangular nonzero in the permuted matrix. Since there are no data dependencies within a level in the permuted matrix (i.e., no edges connecting vertices within a level-set), the $\mathbf{D}_i$ must be diagonal matrices.

With this basic level-set permuted matrix structure, we can use either a forward-looking or backward-looking algorithm. After a diagonal solve determines a set of vector entries $\tilde{\mathbf{x}}_i$, the forward-looking algorithm uses $\tilde{\mathbf{x}}_i$ to immediately update $\tilde{\mathbf{x}}_j, j > i$ with the matrix-vector product operations in the ith column block. A backward-looking algorithm uses all previously computed $\tilde{\mathbf{x}}_i$, $i = 1, \dots, l-1$, in a series of matrix-vector products updates immediately before computing $\tilde{\mathbf{x}}_l$. Both algorithms have the same operation counts but have different memory access patterns for the matrices and vectors. In particular, the forward-looking algorithm exploits the temporal locality of the previously calculated $\tilde{\mathbf{x}}_i$ that are used in the matrix-vector products while the backward-looking algorithm exploits the temporal locality of $\tilde{\mathbf{x}}_i$ that are being determined/stored. While both algorithms have different advantages, we chose to implement the backward-looking algorithm, since we were able to use our compressed row storage matrices in a more natural manner. The operations needed to solve the permuted system for $\tilde{\mathbf{x}}$ in this backward-looking algorithm are as follows

$$
\begin{aligned}
\tilde{\mathbf{x}}_1 &= \mathbf{D}_1^{-1}\tilde{\mathbf{y}}_1 \\
\tilde{\mathbf{x}}_2 &= \mathbf{D}_2^{-1}\left(\tilde{\mathbf{y}}_2 - \mathbf{A}_{2,1}\tilde{\mathbf{x}}_1\right) \\
\vdots \quad &\vdots \quad\quad \vdots \\
\tilde{\mathbf{x}}_l &= \mathbf{D}_l^{-1}\left(\tilde{\mathbf{y}}_l - \mathbf{A}_{l,1}\tilde{x}_1 - \ldots - \mathbf{A}_{l,l-1}\tilde{\mathbf{x}}_{l-1}\right).
\end{aligned}
$$

(Note that the above operations were written to elucidate this algorithm but in practice the sparse matrix-vector product (SpMV) operations for each level can be combined into one SpMV.) $\mathbf{x}$ can be subsequently recovered by the operation $\mathbf{P}^T\tilde{\mathbf{x}}$.

The vector entries at each level (those in $\tilde{\mathbf{x}}_i$ above) can be calculated independently. Thus, in our threaded solver kernel, the computation in each level can be distributed across threads (e.g., with an OpenMP/TBB like **parallel for** operation) without the need for synchronization. However, synchronization is needed between the levels of the algorithm to ensure that the vector entries $\tilde{\mathbf{x}}_j$ needed for matrix-vector product portion of each level of computation (i.e., $\mathbf{A}_{i,j}\tilde{\mathbf{x}}_j$) have been previously determined. In this context, we examine how specific factors affect the performance of this multithreaded triangular solve kernel.

This approach is most beneficial for solving triangular systems resulting from incomplete factorizations, where the resulting matrix factors are sufficiently sparse to yield sufficiently large levels. For matrices that do not result in sufficiently large levels, this approach to parallelism will not be particularly effective (as we will see in the subsequent section). However, for matrices where the resulting levels are sufficiently large, the synchronization costs in our multithreaded algorithm should be small enough to allow for good parallel performance.

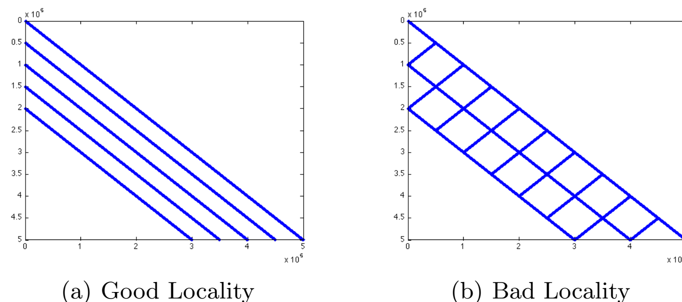
1.3 Related work

Saltz described the usage of directed acyclic graphs and wavefront (level-set) methods for obtaining parallelism for the sparse triangular solve operation in [5]. In this paper, his work focused on sparse triangular systems generated by incomplete factorizations arising from discretization of partial differential equations. This approach was applicable to both shared memory and message passing systems. Rothberg and Gupta addressed the sparse triangular solve bottleneck in the context of the incomplete Cholesky conjugate gradient algorithm ([6]). They argued that the two most promising algorithms at that time (one of which was the level-set method) performed poorly on the more modern shared memory machines that utilized deep memory hierarchies. One of the problems they found was that the level-set algorithm's performance was negatively impacted by the poor spatial locality of the data. This is not a major difficulty for our implementation since we explicitly permute the triangular matrix unlike the original implementation that simply accessed the rows in a permuted order. In more recent work ([7]), Mayer has developed two new algorithms for solving triangular systems on shared memory architectures. The first algorithm uses a block

partitioning to split the triangular matrix across both rows and columns in a 2D Cartesian manner. Given this partitioning, the operations using blocks on the same anti-diagonal can be computed in parallel on different threads. The difficulty with this method is finding a good partitioning of blocks that balances the work. The second method is a hybrid method combining the first method with a method of solving by column blocks (after a diagonal block has been solved, the updates in the column block below may be done in parallel). The hybrid method is easier to partition than the first method. Mayer's results were somewhat modest in the speedups that were obtained. However, the methods are more general than the level-set method and may be effective over a larger range of matrices.

2 Factors Affecting Performance

One factor we examine is data locality for the matrices. First, we experiment on two special types of matrices (Figure 2), where the number of rows is the same for each level and the matrices are already ordered by level-sets. One of these special matrices results in good data locality (Figure 2(a)) in that threads will not use vector entries that another thread has it computed in its computation. This can be seen in the precisely placed off-diagonal bands in these matrices that ensure that if a vector entry x_i is calculated by a thread, any subsequent computation involving x_i (i.e., computation corresponding to nonzeros in column i) will assigned to that same thread. We can enforce this good data locality since these matrices have the same number of rows per level-set and we assign row operations to threads in a contiguous block manner. The other matrix results in bad data locality (Figure 2(b)) in that threads will often use vector entries calculated by another thread. Again, this can be seen in the precisely placed off-diagonal nonzero blocks in these matrices that ensure that if a vector entry x_i is calculated by a thread, subsequent computation involving x_i in the next level will not be assigned to that same thread.



(a) Good Locality (b) Bad Locality

Fig. 2. Nonzero patterns for matrix types.

We also look at variants of the triangular solve algorithm with different barriers and thread affinity settings. The barrier is an important part of this level set method, providing synchronization between the levels of our triangular solve. The first type of barrier is somewhat passive and uses mutexes and conditional wait statements. All the threads wait (**pthread_cond_wait**) until every thread has entered the barrier. Then a signal is broadcast (**pthread_cond_broadcast**) that allows the threads to exit the barrier. The disadvantage with this more passive barrier is a thread calling this barrier might be switched to a different computational core while waiting for a signal in the conditional wait statement. The second type of barrier is more active and uses spin locks and active polling. A thread entering this barrier will actively poll until all threads have reached the barrier. This makes it less likely for the threads to be switched, which is a good thing assuming there is nothing else being computed simultaneously with the triangular solve. Thread affinity describes how likely a thread is to run on a particular core. By setting the thread affinity, we can bind a thread to a particular core, which can be beneficial to the performance of numerical kernels. This also allows us to ensure our threads are running on the same socket on machines with multiple sockets. This may be desirable for numerical algorithm, especially if there is effective utilization of a cache shared between the cores on a socket (e.g., L3 cache on Nehalem). When setting the thread affinity, we set the affinity for each thread to a different core on the same socket.

3 Numerical Experiments

We implemented a level set triangular solve prototype that solves triangular systems of ten levels, with the same number of rows for each level. For this prototype, the rows in a level are distributed in a block fashion to different threads that will perform the computation on those rows. This simple set up allows us to easily control the factor of data locality. We experiment on a range of different size matrices and run our experiments on one, two, and four threads to study the scalability of the algorithm variants.

We have performed these experiments on two different multi-core systems. The first system is an Intel Nehalem system running Linux with a two socket motherboard with 2.93 GHz quad-core Intel Xeon processors for a total of 8 cores. Intel's Turbo Boost Technology is turned off on this system, so two threads should run at the same clock speed as one thread. The second system is an AMD Istanbul system running Linux with a two socket motherboard with 2.6 GHz six-core AMD Opteron processors for a total of 12 cores.

3.1 Barriers

First, we compare the results for the different types of barriers. Figures 3 and 4 show results for the triangular solves on the good data locality matrices of various sizes when the thread affinity is set. For the Nehalem system, we show results for 2, 4, and 8 threads. For the Istanbul system, we show results for 2, 6,

and 12 threads. Parallel speedups are presented for both the active and passive barrier variants.

For both the Nehalem and the Istanbul systems, it is clear that the active barrier is necessary to obtain good scalability, especially for the smaller sized matrices and runs with many threads. Figure 5 shows a runtime comparison of the two implementations using different barriers for two of the matrices (bad data locality and thread affinity on) for 1, 2, and 4 threads. It is clear from both the Nehalem and Istanbul plots that having an active barrier is important for scalability.

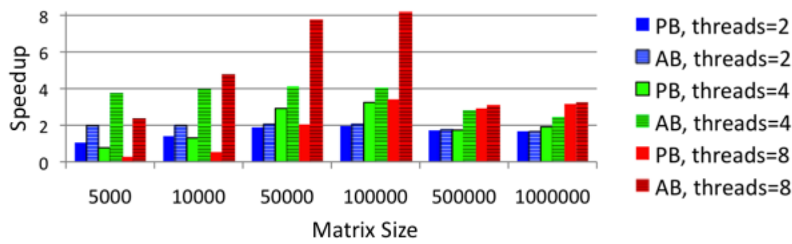


Fig. 3. Effects of different barriers on Nehalem system: thread affinity on, good data locality. Speedups for passive barrier (PB) and active barrier (AB) for 2, 4, and 8 threads.

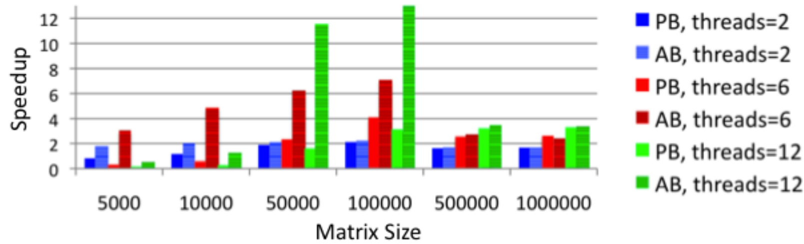
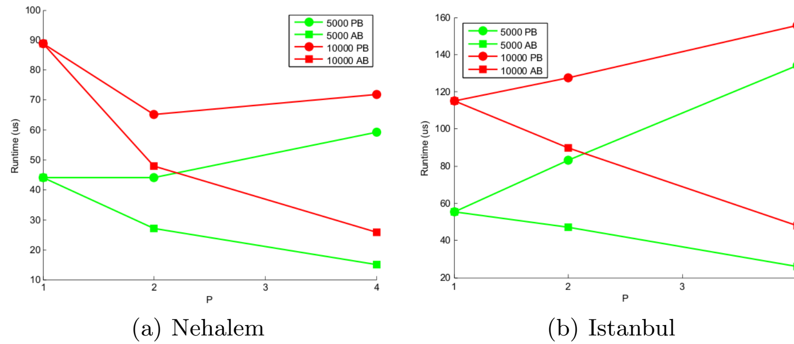


Fig. 4. Effects of different barriers on Istanbul system: thread affinity on, good data locality. Speedups for passive barrier (PB) and active barrier (AB) for 2, 6, and 12 threads.

3.2 Thread affinity

Next, we examine the effects of thread affinity on the scalability of our triangular solve algorithm. Figures 6 and 7 show results for the triangular solves on the



(a) Nehalem (b) Istanbul

Fig. 5. Effects of different barriers: sizes 5000 and 10000, thread affinity on, bad data locality. Runtimes for passive barrier (PB) and active barrier (AB).

good data locality matrices of various sizes with the active barrier. For both the Nehalem and the Istanbul systems, it is clear that the thread affinity is not as important of a factor in scalability as the barrier type. However, for some of the smaller data sizes, setting the thread affinity does seem to improve the scalability. Figure 8 shows the runtimes for two matrices (bad data locality and active barrier) with thread affinity on or off for 1, 2, and 4 threads. It seems that thread affinity is somewhat important for these problem sizes, especially on the Nehalem system.

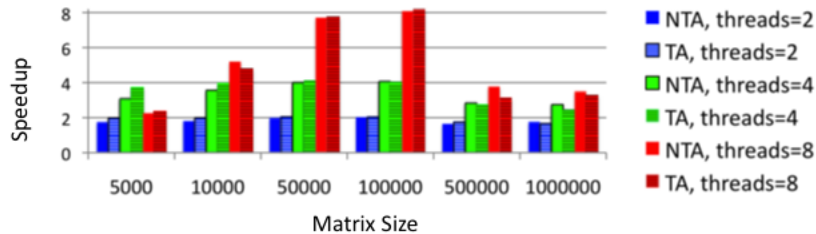


Fig. 6. Effects of binding threads to cores by setting thread affinity on Nehalem system: active barrier, good data locality. Speedups for algorithm when setting thread affinity (TA) and not setting thread affinity (NTA) for 2, 4, and 8 threads.

3.3 Data locality

Finally, we examine the impact of data locality on the scalability of the triangular solves. Figure 9 shows a comparison between the results of the two different types of matrices (one with good locality and the other with bad data locality) of size

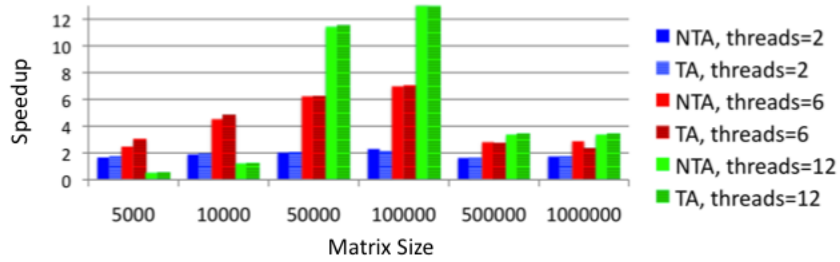
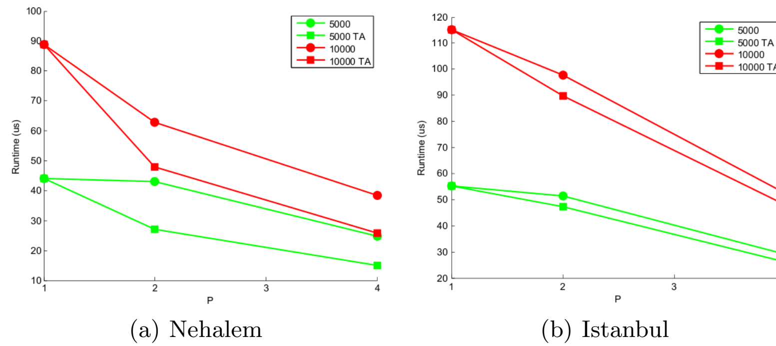


Fig. 7. Effects of binding threads to cores by setting thread affinity on Istanbul system: active barrier, good data locality. Speedups for algorithm when setting thread affinity (TA) and not setting thread affinity (NTA) for 2, 6, and 12 threads.



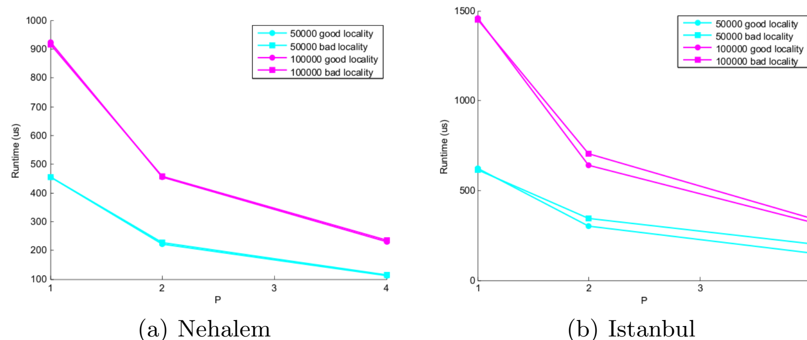
(a) Nehalem (b) Istanbul

Fig. 8. Effects of setting thread affinity: sizes 5000 and 10000, active barrier, bad data locality. Runtimes for setting thread afinity (TA) and not setting thread affinity.

50000 and 100000 rows. We see basically no difference for these types of matrices for Nehalem and only a very slight difference for Istanbul. The results for these two sizes was typically of what we observed overall.

3.4 More realistic problems

In the previous subsections, we solved triangular systems for a very specific set of matrices. These matrices were designed to have a specific structure that allowed us to study the importance of data locality in a very simple environment. These matrices were sufficient to get a good handle of the factors affecting performance in an ideal scenario. In this subsection, we study the impact of barrier type and thread affinity in more realistic situation, solving triangular systems resulting from four symmetric matrices obtained from the University of Florida Sparse Matrix Collection [8]. These four matrices are shown in Table 1 with their respective number of rows, number of nonzeros, and application areas. We generalized the prototype solver used in the previous subsections to calculate



(a) Nehalem (b) Istanbul

Fig. 9. Effects of data locality: active barrier, thread affinity on. Runtimes for good and bad data locality matrices (sizes 50000 and 100000).

level sets, permute the matrices, and solve the triangular system for any lower triangular system.

Table 1. Symmetric Matrix Info

Name	N	nnz	N/nlevels	application area
asic680ks	682,712	2,329,176	13932.9	circuit simulation
cage12	130,228	2,032,536	1973.2	DNA electrophoresis
pkustk04	55,590	4,218,660	149.4	structural engineering
bcsstk32	44,609	2,014,701	15.1	structural engineering

We take the lower triangular part of the matrices shown in Table 1 to be our lower triangular matrices (i.e., zero fill incomplete factorizations). The fourth column of Table 1 gives the average number of rows per level for the level-sets determined from the lower triangular part of these matrices. We picked these four matrices deliberately to cover a range for this statistic. As we did with the simple matrices of the previous subsections, we compare the results for the active and passive barrier variants. Figures 10 and 11 show results for the triangular solves when the thread affinity is set, comparing the two barrier types.

Again we see for both the Nehalem and the Istanbul systems that the active barrier is necessary to obtain good scalability. The difference is particularly striking for the larger numbers of threads. As expected, the solves scaled better when the matrices had large numbers of rows per level. In particular, the **asic680ks** and **cage12** matrices, which had the largest numbers of rows per level, scaled very well (especially on the Istanbul architecture). However, the **bcsstk32** matrix, which had approximately 15.1 rows per level, actually required more runtime as the number of threads increased. This is not too surprising since with an

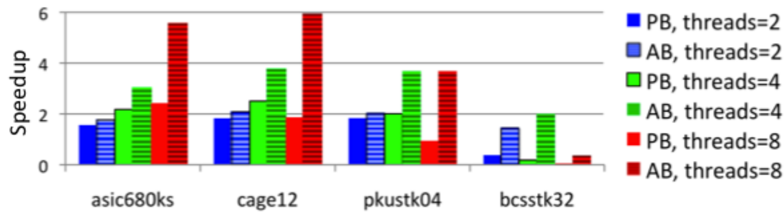


Fig. 10. Application matrices. Effects of different barriers on Nehalem system: thread affinity on. Speedups for passive barrier (PB) and active barrier (AB) for 2, 4, and 8 threads.

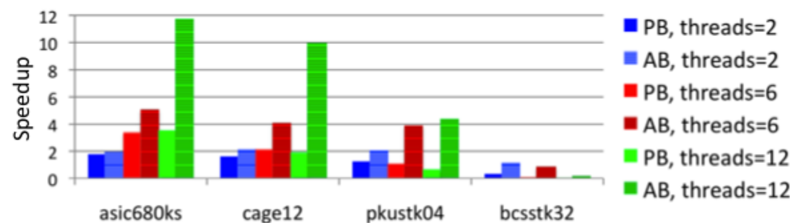


Fig. 11. Application matrices. Effects of different barriers on Istanbul system: thread affinity on. Speedups for passive barrier (PB) and active barrier (AB) for 2, 6, and 12 threads.

average of 15.1 rows per level, many levels would not have one row per thread, let alone provide enough work to amortize the cost of the synchronization step.

Again, we examine the effects of thread affinity on the scalability of our triangular solve algorithm. Figures 12 and 13 show results for the triangular solves for these more realistic matrices with the active barrier. For both the Nehalem and the Istanbul systems, it is clear that the thread affinity is not as important of a factor in scalability as the barrier type. For several problems, we see a slight increase in speedup when thread affinity is on. However, there are several counter examples where the speedup slightly decreases.

4 Summary and Conclusions

In pursuit of more scalable solvers that scale to hundreds of thousands of computational cores on multi-core architectures, we are researching hybrid MPI/threaded algorithms that should lower iterations counts by reducing the number of MPI tasks (and subdomains). An essential part of these algorithms are scalable threaded numerical kernels such as the triangular solver on which we focused. We examined three different factors that affect the performance of the threaded level-set triangular solver. Of these three factors, the barrier type was shown to have the most impact, with an active barrier greatly increasing the parallel performance

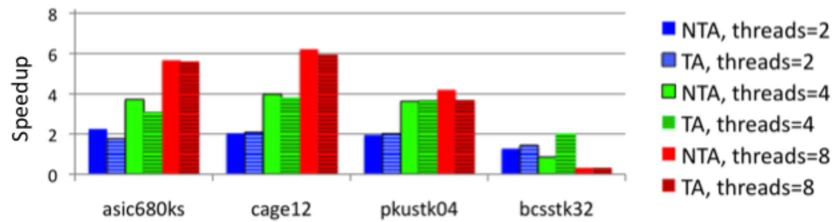


Fig. 12. Application matrices. Effects of binding threads to cores by setting thread affinity on Nehalem system: active barrier. Speedups for algorithm when setting thread affinity (TA) and not setting thread affinity (NTA) for 2, 4, and 8 threads.

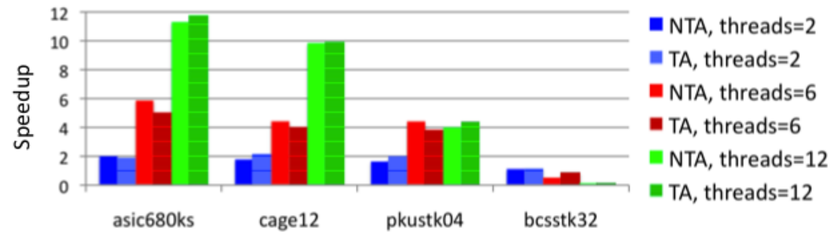


Fig. 13. Application matrices. Effects of binding threads to cores by setting thread affinity on Istanbul system: active barrier. Speedups for algorithm when setting thread affinity (TA) and not setting thread affinity (NTA) for 2, 6, and 12 threads.

when compared to a more passive barrier. Although it is not always be possible (e.g., if additional computation takes place on the same cores concurrently), we advocate using as aggressive of a barrier as possible in this type of algorithm. Our results showed that binding the threads to processor cores had less impact than the barrier type. However, it did improve the performance for some cases and may be a reasonable approach to take for multithreaded numerical kernels where the number of active threads is not more than the number of computational cores. With an active barrier and thread binding to cores, we were able to achieve excellent parallel performance for a range of matrix sizes for the ideal matrices as well as three of the four more realistic matrices that we studied.

We also examined the impact of data locality on the scalability of the triangular solves, comparing a matrices with good and bad data locality. It is unclear from our results whether data locality is an important factor in the parallel performance. It is possible that our bad data locality matrices do not have poor enough data locality to see a very large effect. It is also possible that our matrices are too sparse and that we would see more of an effect for denser matrices. But perhaps the memory systems are too fast for the locality of the data in these sparse matrix triangular solves to greatly impact the scalability of the algorithm.

If the data locality becomes an issue more general classes of triangular matrices, we would need to explore ordering techniques to mitigate this problem.

Of more importance was the sparsity structure of the matrices and how this sparsity translated into level-sets. This was apparent in our study that utilized symmetric matrices obtained from various application areas. We saw a strong correlation between parallel performance of our multithreaded triangular solver and the average number of rows per level in the level-set permuted matrices. The matrix obtained from the **bcsstk32** matrix resulted in only 15.1 rows per level. The solution of these system actually slowed down as threads were added. The lower triangular part of a tridiagonal matrix would be the worse case with a linear DAG and only 1 row per level. Clearly for these types of matrices, the level-set method is not scalable. Perhaps a feasible approach is to calculate this statistic in the DAG analysis phase to determine whether or not to use the level-set algorithm.

Acknowledgments

We thank Chris Baker, Cédric Chevalier, Karen Devine, Doug Doerfler, Paul Lin, and Kevin Pedretti for their input and many helpful discussions. This work was funded as part of the Extreme-scale Algorithms and Software Institute (EASI) by the Department of Energy, Office of Science. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration, under contract DE-AC-94AL85000.

References

1. Lin, P., Shadid, J., Sala, M., Tuminaro, R., Hennigan, G., Hoekstra, R.: Performance of a parallel algebraic multilevel preconditioner for stabilized finite element semiconductor device modeling. Journal of Computational Physics **228**(17) (2009) 6250–6267
2. Hennigan, G., Hoekstra, R., Castro, J., Fixel, D., Shadid, J.: Simulation of neutron radiation damage in silicon semiconductor devices. Technical Report SAND2007-7157, Sandia National Laboratories (2007)
3. Lin, P.T., Shadid, J.N.: Performance of an MPI-only semiconductor device simulator on a quad socket/quad core InfiniBand platform. Technical Report SAND2009-0179, Sandia National Laboratories (2009)
4. Li, X.S., Shao, M., Yamazaki, I., Ng, E.G.: Factorization-based sparse solvers and preconditioners. Journal of Physics: Conference Series **180**(1) (2009) 012015
5. Saltz, J.H.: Aggregation methods for solving sparse triangular systems on multiprocessors. SIAM Journal on Scientific and Statistical Computing **11**(1) (1990) 123–144
6. Rothberg, E., Gupta, A.: Parallel iccg on a hierarchical memory multiprocessor – addressing the triangular solve bottleneck. Parallel Computing **18**(7) (1992) 719–741
7. Mayer, J.: Parallel algorithms for solving linear systems with sparse triangular matrices. Computing **86**(4) (2009) 291–312

8. Davis, T.A. The University of Florida Sparse Matrix Collection (1994) Matrices found at http://www.cise.ufl.edu/research/sparse/matrices/.