Load balancing for Regular Meshes on SMPs with MPI

Vivek Kale and William Gropp

University of Illinois at Urbana-Champaign, IL, USA,
{vivek,wgropp}@illinois.edu

Abstract. Domain decomposition for regular meshes on parallel computers has traditionally been performed by attempting to exactly partition the work among the available processors (now cores). However, these strategies often do not consider the inherent system noise which can hinder MPI application scalability to emerging peta-scale machines with 10000+ nodes. In this work, we suggest a solution that uses a tunable hybrid static/dynamic scheduling strategy that can be incorporated into current MPI implementations of mesh codes. By applying this strategy to a 3D jacobi algorithm, we achieve performance gains of at least 16% for 64 SMP nodes.

1 Introduction

Much literature has emphasized effective decomposition strategies for good parallelism across nodes of a cluster. Recent work with hybrid programming models for clusters of SMPs has often focused on determining the best split of threads and processes, and the shape of the domains used by each thread [1, 2]. In fact, these static decompositions are often auto-tuned for specific architectures to achieve reasonable performance gains. However, the fundamental problem is that this "static scheduling" assumes that the user's program has total access to all of the cores all of the time; these static decomposition strategies cannot be tuned easily to adapt in real-time to system noise (particularly due to OS jitter). The occassional use of the processor cores, by OS processes, runtime helper threads, or similar background processes, introduce noise that makes such static partitioning inefficient on a large number of nodes. For applications running on a single node, the general system noise is small though noticeable. Yet, for next-generation peta-scale machines, improving mesh computations to handle such system noise is a high priority. Current operating systems running on nodes of high-performance clusters of SMPs have been designed to minimally interfere with these computationally intensive applications running on SMP nodes [3], but the small performance variations due to system noise can still potentially impact scalability of an MPI application for a cluster on the order of 10,000 nodes. Indeed, to eliminate the effects of process migration, the use of approaches such as binding compute threads/processes to cores, just before running the application, is advocated [3]. However, this only provides a solution for migration and neglects overhead due to other types of system noise.

In this work, we illuminate how the occassional use of the processor cores by OS processes, runtime helper threads, or similar background processes, introduce noise that makes such static schedules inefficient. In order to performance tune these codes with system noise in mind, we propose a solution which involves a partially dynamic scheduling strategy of work. Our solution uses ideas from task stealing and work queues to dynamically schedule tasklets. In this way, our MPI mesh codes work with the operating system running on an SMP node, rather than in isolation from it.

2 Problem Description

Our model problem is an exemplar of regular mesh code. For simplicity, we will call it a Jacobi algorithm, as the work that we perform in our model problem is the Jacobi relaxation iteration in solving a Poisson problem. However, the data and computational pattern are similar for both regular mesh codes (both implicit and explicit) and for algorithms that attempt to evenly divide work among processor cores (such as most sparse matrix-vector multiply implementations). Many MPI implmentations of regular mesh codes traditionally have a predefined domain decomposition, as can be seen in many libraries and microbenchmark suites [4]. This optimal decomposition is necessary to reduce communication overhead, minimize cache misses, and ensure data locality. In this work, we consider a slab decomposition of a 3-dimensional block implemented in MPI/pthreads hybrid model, an increasingly popular model for taking advantage of clusters of SMPs.

We use a problem size and dimension that can highlight many of the issues we see in real-world applications with mesh computations implemented in MPI: specifically, we use a 3D *block* with dimensions $64 \times 512 \times 64$ on each node for a fixed 1000 iterations. For our 7-point stencil computation, this generates a total of 1.6 GFLOPS per node With this problem size, we can ensure that computations are done out-of-cache so that it is just enough to excercise the full memory hierarchy. The block is partitioned into *vertical slabs* across processes along the X dimension. Each vertical slab is further partitioned into *horizontal slabs* across threads along the Y dimension. Each vertical slab contains a static section(top) and a dynamic section(bottom). We use this decomposition strategy because of its simplicity to implement and tune different parameters in our search space. A MPI *border exchange* communication occurs between left and right borders of blocks of each process across the YZ planes. The border exchange operation uses `MPI_Isend` and `MPI_Irecv` pair, along with an `MPI_Waitall`.We mitigate the issue of first-touch as noted in [5] by doing parallel memory allocation during the initialization of our mesh.

For such regular mesh computations, the communication between processes, even in an explicit mesh sweep, provides a synchronization between the processes. Any load imbalance between the processes can be amplified, even when using a good (but static) domain decomposition strategy. If even 1% of nodes are affected by system interference during one iteration of a computationally intensive MPI application on a cluster with 1000s of nodes, several nodes will be affected by noise during each iteration. Our solution to this problem is to use a partially dynamic scheduling strategy, and is presented in the section that follows.

3 Performance Tuning Experimentation

3.1 Performance Tuning Technique

The technique for supporting dynamic scheduling of computation was implemented with a queue that was shared among threads. Each element of the shared queue (we refer to it as a *tasklet*) contains the specification of what work the thread executing this tasklet is responsible for, and a flag indicating whether the tasklet has been completed by a thread. In order to preserve locality (so that in repeated computations the same threads can get the same work), we also maintain an additional tag specifying the last thread that ran this tasklet. In the execution of each iteration of the Jacobi algorithm, there are 3 distinct phases: MPI communication, statically scheduled computation, and dynamically scheduled computation. In phase 1, thread 0 does the MPI communication for border exchange. During this time, all other threads must wait at a thread barrier. In phase 2, a thread does all work that is statically allocated to it. Once a thread completes its statically allocated work it immediately moves to phase 3, where it starts pulling the next available tasklet from the queue shared among other threads, until the queue is empty. As in the completely static scheduled case, after threads have finished computation, they will need to wait at a barrier before continuing to the next iteration. The percentage of dynamic work, granularity/number of tasklets, and number of queues for a node, is specified as parameter. Through our experimental studies of tuning our dynamic scheduling strategy, we pose the following questions:

1. Does partially dynamic scheduling improve performance for mesh computations that have traditionally been completely statically scheduled?
2. What is the tasklet granularity we need to use for maintaining load balance of tasklets across threads?
3. In using such a technique, how can we decrease the overheads of synchronization of the work queues used for dynamic scheduling?
4. What is the impact of the technique for scaling to many nodes?

In the sections that follow, we first demonstrate the benefits of partial dynamic scheduling on one node in 3.2. Section 3.3 describes the effect of task granularity. Section 3.4 examines the impact on MPI runs with multiple nodes. Our experiments were conducted on a system with Power575 SMP nodes with 16 cores per node, and the operating system was IBM AIX. We assign a compute thread to each core, ensuring that the node is fully subscribed (ignoring the 2-way SMT available on these nodes as there are only 16 sets of functional units). If any OS or runtime threads need to run, they must take time away from one of our computational threads.

3.2 Reducing impact of OS jitter: Dynamic vs Static scheduling

As mentioned above, threads first complete all static work assigned to it. Once a thread completes this stage, it moves to the dynamic phase, where it dequeues tasklets from the task queue. In the context of the stencil computation experimentation we do, each thread is assigned a horizontal slab from the static region at compile time. After a thread fully completes its statically allocated slab, it completes as many tasklets of the dynamic region as it can. The number of tasklets is a user-specified parameter. To explore the impact of using dynamic scheduling with locality preference, we enumerate 4 separate cases, based on the dynamic scheduling strategy.

1. 0% dynamic: Slabs are evenly partitioned, with each thread being assigned one slab. All slabs are assigned to threads at compile-time.
2. 100% dynamic + no locality: All slabs are dynamically assigned to threads via a queue.
3. 100% dynamic + locality: Same as 2, except that when a thread tries to dequeue a tasklet, it first searches for tasklets that it last executed in a previous jacobi iteration.
4. 50% static, 50% dynamic + locality: Each thread first does its static section, and then immediately starts pulling tasklets from the shared work queue. This approach is motivated by a desire to reduce overhead in managing the assignment of tasks to cores.

For the cases involving dynamic scheduling, we initially assume the number of tasklets to be 32, and that all threads within an MPI process share one work queue. We preset the number of iterations to be 1000 (rather than using convergence criteria) to allow us to more easily verify our results. In our experiments, we choose 1000 iterations as this adequately captures the periodicity of the jitter induced by the system services during a trial [3]. Figure 1 below shows the average performance we obtained over 40 trials for each of these cases. From the figure, we can see that the 50% dynamic scheduling gives significant performance benefits over the traditional static scheduling scheduling case. Using static scheduling, the average execution time we measure was about 7.00 seconds of wall-clock time. We make note that of the 40 trials we did, we obtained 6 *lucky* runs where the best performance we got was in the range 6.00 - 6.50 seconds. The remaining 34 runs were between 7.00 - 8.00 seconds. Using fully dynamic scheduling with no locality, performance was slightly worse than the statically scheduled case. For this case, there were some small performance variations (within 0.2 seconds) across the 40 trials; these were most probably due to the varying number of cache misses, in addition to system service interference. Using locality with fully dynamic scheduling, the performance variations over 40 trials here were even lower (within 0.1 seconds). Using the 50% dynamic scheduling strategy, the execution time was 6.53 seconds, giving us over 7% performance gain over our baseline static scheduling. Thus, we notice that just by using a reasonable partially dynamic scheduling strategy, performance variation can be reduced and overall performance can be improved. In all cases using dy-

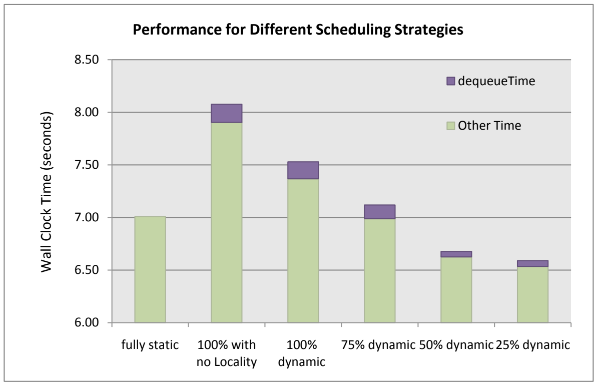

Fig. 1. The performance of different scheduling strategies used with the Jacobi Computation with 64 by 512 by 64 size block.

namically scheduling, the thread idle times(not shown here) contribute to the largest percentage overhead. The high overhead in case 2 is likely attributed to the fact that threads suffer from doing non-local work. Because some threads suffer cache misses while others do not, the overall thread idle time (due to threads waiting at barriers) could be particularly high.

3.3 Tuning Tasklet Granularity for Reduced Thread Idle Time

As we noticed in the previous section, the idle times account for a large percentage of the performance. Total thread idle time (summed across threads) can be high because of load imbalance. Our setup above used 32 tasklets. However, the tasklets may have been too coarse grained (each tasklet has a 16-plane slab). With very coarse granularity, performance suffers because threads must wait (remain idle) at a barrier, until all threads have completed their dynamic phase. As a first strategy, we varied the number of tasklets, using 16, 32, 64, 96, and 128 tasklets as our test cases. The second strategy, called skewed workloads, addresses the tradeoff between fine-grain tasklets and coarse-grain tasklets. In this strategy, we use a work queue containing variable sized tasklets, with larger tasklets at the front of the queue and smaller tasklets towards the end. Skewed workloads reduce the contention overhead for dequeuing tasklets (seen when using fine-granularity tasklets) and also reduce the idle time of threads (seen when using coarse-grain tasklets). In figure 2, we notice that as we increase number of tasklets from 16 to 64 tasklets (decreasing tasklet size) we obtain significant performance gains, and the gains come primarily from the reduction in idle times. Overall, we notice that the performance increases rapidly in this region. As we increase from 64 to 128 tasklets, performance starts to decrease, primarily due to the contention for the tasklets and the increased dequeue overhead. We also see that performance of the skewed strategy (especially with 50% dynamic scheduling) is comparable to that of 64 tasklets, which has the best performance. In this way, a skewed strategy can yield competitive performance without needing

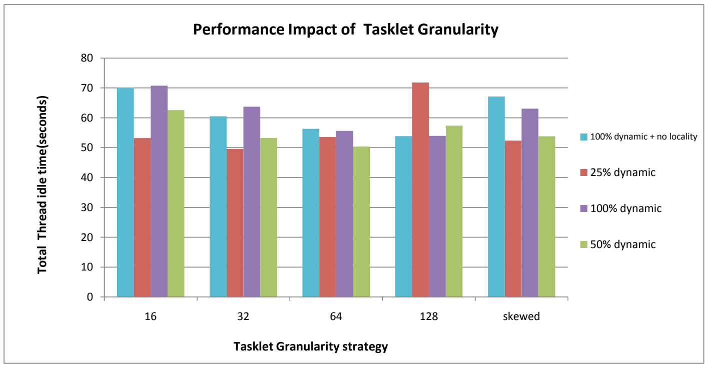

Fig. 2. Increasing task granularity to helps improve performance, particularly because of reduced thread idle times. However, at 128 tasklets the performance starts to degrade due to increasing contention for tasklets.

to predefine the tasklet granularity. To understand how tuning with a skewed workload benefits performance, figure 3 shows the distribution of timings for each of the 1000 iterations of the jacobi algorithm, comparing between static scheduling, 50% dynamic scheduling with fixed size tasklets, and 50% dynamic scheduling with skewed workloads. Using static scheduling, the maximum iteration time was 9.5 milliseconds(ms), about 40% larger than the average time of all iterations. Also, the timing distribution is bimodal, showing that half the iterations ran optimally as tuned to the architecture(running in about 6 ms), while the other half were slowed down by system noise(running in about 7.75 ms). Using 50% dynamic scheduling, the maximum iteration time is reduced to 8.25 ms, but it still suffers due to dequeue overheads, as can be seen by the mean of 7.25 ms. By using a skewed workload strategy, we see that the max is also 8.25 ms. However, the mean is lower (6.75 ms) than that seen when using fixed size tasklets, because of the lower dequeue overhead that this scheme provides. The skewed workloads provided 7% performance gains over the simple 50% dynamic scheduling strategy, which uses fixed-size coarse-grain tasklets of size 32. Furthermore, the reduced max time when using dynamic scheduling indicates that our dynamic scheduling strategy better withstands perturbations caused by system noise.

3.4 Using our technique to improve Scalability

For many large MPI applications (especially with barriers) running on many nodes of a cluster, even a small system service interruption on a core of a node can accumulate to offset the entire computation, and degrade performance. In this way, the impact of a small load imbalance across cores is amplified for a large number of processes. This reduces the ability for application scalability, particularly for a cluster with a very large number of nodes (and there are many machines with more than 10000 nodes). To understand how our technique can be used to improve scalability, we tested our skewed workload with a 50% dynamic

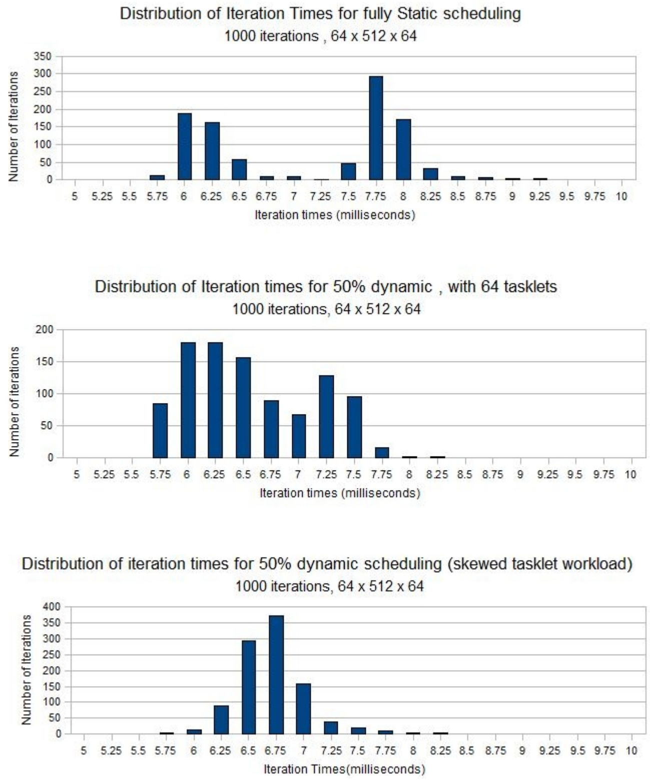

Fig. 3. Histogram view showing the performance variation of iterations for static scheduling, 50% dynamic scheduling with fixed-size tasklet granularity, and 50% dynamic scheduling with skewed workload strategy.

scheduling strategy on 1, 2, 4, 8, 16, 32, and 64 nodes of a cluster. One core of a node was assigned as a message thread to invoke MPI communication (for border exchanges) across nodes. We used the hybrid MPI/pthread programming model for implementation. Figure 4 shows how as we increase the number of nodes, using 50% dynamic scheduling always outperforms the other strategies and scales well. At 64 nodes, the 50% dynamic scheduling gives us on average a 30% performance improvement over the static scheduled case. As we can see for the case with static scheduling, a small overhead due to system services is amplified at 2 nodes and further degrades as we move up to 64 nodes. In contrast, for the 50% dynamic scheduling strategy using skewed workloads, the performance does not suffer as much when increasing the number of nodes, and our jitter mitigation techniques' benefits are visible at 64 nodes. To see the reasons for better scalability, we consider the iteration time distributions for 1 node in our 64 node runs, as shown in figure 5 (the distributions across all nodes were roughly the same). Compared to the top left histogram of figure 3, the histogram in figure 5 shows

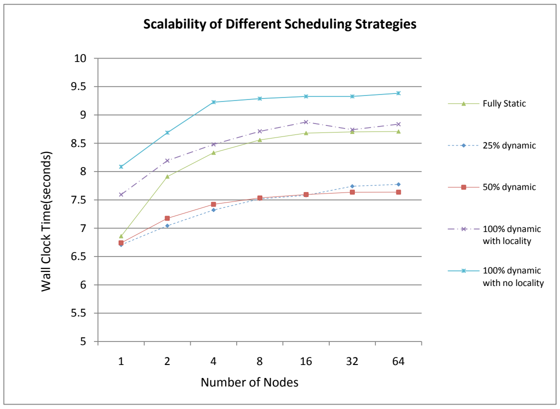

Fig. 4. Scalability results show that the 50% dynamic scheduling strategy performs better and also scales well compared to the traditional static scheduling approach.

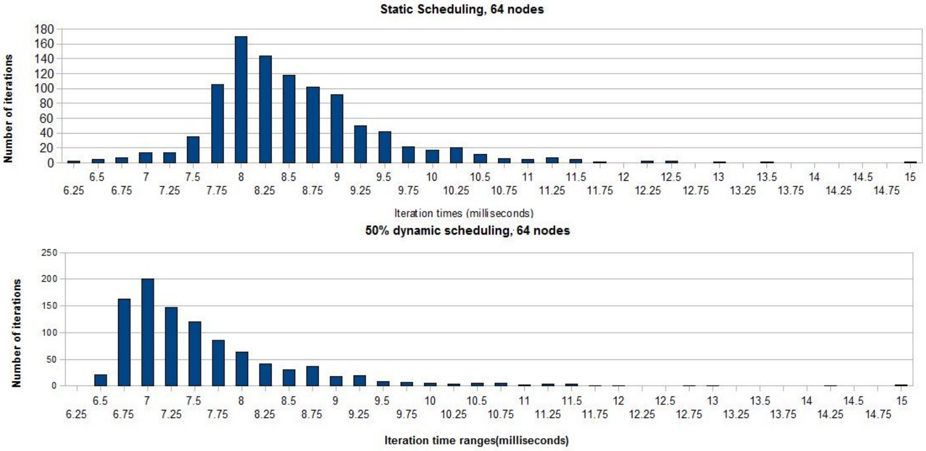

Fig. 5. The histograms (taken from node 0) in a 64 node run are shown. The left histogram corresponds to the static scheduling technique, while the right histogram corresponds to the 50% dynamic scheduling technique.

that the distribution has shifted significantly to the right for static scheduling. This makes sense since each node's jitter occurs at different times. The chain of dependencies through MPI messaging for border exchanges compounds the delay across nodes in consecutive iterations.With dynamic scheduling, the distribution has not shifted as much. For example, the mode(the tallest line) only shifted from 6.75 ms to 7.00 ms. This is because in each iteration, the node that experiences noise mitigates its effect by scheduling delayed tasklets to its other threads.

4 Related Work

The work by [1,5] shows how regular mesh (stencil) codes can be auto-tuned onto a multi-core architecture by enumerating different parameters and using sophisticated machine learning techniques to search for the best parameter configurations. In their work, the search space is based on the architectural parameters. In our work, we suggest another issue that one should be aware of for tuning codes: the random system noise incurred by OS-level events.

Cilk is a programming library [6] intended to enhance performance of many multi-core applications, and uses the ideas of shared queues and work stealing to dynamically schedule work. While the implementation of our dynamic strategy is similar to the Cilk dynamic scheduling strategy, we propose using a dynamic scheduling strategy for just the last fraction of the computation, rather than all of it. Furthermore, our method is locality-aware and allows one to tune this fraction of dynamic scheduling to the inherent system noise. We believe this can be particularly beneficial to scientific codes that are already optimally partitioned across nodes and tuned for the architecture. In [7] dynamic task scheduling with variable task sizes is used as a method for optimizing ScaLaPack libraries. Our work uses predefined, but tuned, task sizes that mitigate the system noise, without incurring dynamic scheduling overhead.

The work in [8] identifies, quantifies, and mitigates sources of OS jitter mitigation sources on large supercomputer. This work suggests different methodologies for handling each type of jitter source. This study suggests primarily modifying the operating system kernel to mitigate system noise. Specific methods for binding threads to cores [9] have been shown to have effect in reducing system interference (particularly process migration and its effects on cache misses) for high-performance scientific codes. However, these approaches cannot mitigate all system noise such as background processes or periodic OS timers. Our approach involves tuning an MPI application to any system noise, rather than modifying the operating system kernel to reduce its interference. In addition, the techniques we present can be used in conjunction with thread binding or other such techniques, rather than as an alternative.

5 Conclusions and Future Work

In this work, we introduced a dynamic scheduling strategy that can be used to improve scalability of MPI implementations of regular meshes. To do this, we started with a pthread mesh code that was tuned to the architecture of a 16-core SMP node. We then incorporated our partially dynamic scheduling strategy into the mesh code to handle inherent system noise. With this, we tuned our scheduling strategy further, particularly considering the grain size of the dynamic tasklets in our work queue. Finally, we added MPI for communication across nodes and demostrated the scalability of our approach. Through proper tuning, we showed that our methodology can provide good load balance and scale to a large number of nodes of our cluster of SMPs, even in the presence of system noise.

For future work, we plan to apply our technique to larger applications such as MILC [4]. We will also incorporate more tuning parameters (we are currently examining more sophisticated work-stealing techniques). In addition, we will tune our strategy so that it works alongside other architectural tuning strategies and other basic jitter mitigation techniques. We also plan to test on clusters with different system noise characteristics. With this, we hope to develop auto-tuning methods for such MPI/pthread code in the search space we have presented.

Acknowledgments

We thank Franck Cappello, as well as students at INRIA, for sharing their experience with OS jitter on IBM Power-series machines. Parts of this research are part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

References

1. Williams, S., Carter, J., Oliker, L., Shalf, J., Yelick, K.A.: Optimization of a lattice Boltzmann computation on state-of-the-art multicore platforms. Journal of Parallel and Distributed Computing (2009)
2. Cappello, F., Etiemble, D.: MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In: Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM), Washington, DC, USA, IEEE Computer Society (2000)
3. Mann, P.D.V., Mittaly, U.: Handling OS jitter on multicore multithreaded systems. In: IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society (2009)
4. Shi, G., Kindratenko, V., Gottlieb, S.: The bottom-up implementation of one MILC lattice QCD application on the Cell blade. International Journal of Parallel Programming **37** (2009)
5. Kamil, S., Chan, C., Williams, S., Oliker, L., Shalf, J., Howison, M., Bethel, E.W.: A generalized framework for auto-tuning stencil computations. In: In Proceedings of the Cray User Group Conference. (2009)
6. Blumofe, R.D., Joerg, C.F., Kuszmaul, B.C., Leiserson, C.E., Randall, K.H., Zhou, Y.: Cilk: An efficient multithreaded runtime system. In: Journal of Parallel and Distributed Computing. (1995)
7. Song, F., YarKhan, A., Dongarra, J.: Dynamic task scheduling for linear algebra algorithms on distributed-memory multicore systems. In: SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, New York, NY, USA, ACM (2009)
8. Petrini, F., Kerbyson, D.J., Pakin, S.: The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of ASCI Q. In: SC '03: Proceedings of the 2003 ACM/IEEE conference on Supercomputing, Washington, DC, USA, IEEE Computer Society (2003)
9. Klug, T., Ott, M., Weidendorfer, J., Trinitis, C., Mnchen, T.U.: autopin, automated optimization of thread-to-core pinning on multicore systems (2008)