# Experimenting with low-overhead OpenMP runtime on IBM Blue Gene/Q

A. E. Eichenberger
K. O'Brien

*As newer supercomputers continue to increase the number of threads, there is growing pressure on applications to exploit more of the available parallelism in their codes, including coarse-, medium-, and fine-grain parallelism. OpenMP™ is one of the dominant shared-memory programming models and is well suited for exploiting medium- and fine-grain parallelism. OpenMP research has focused on application tuning, compiler optimizations, programming-model extensions, and porting to distributed-memory platforms; however, we have found that current algorithms used to implement basic OpenMP constructs have significant overheads and scale poorly. In this paper, we explore low-overhead, scalable algorithms for creating parallel regions and demonstrate reductions in overhead of up to a factor of 5 on an IBM Blue Gene®/Q node.*

## Introduction

Current trends in computer architecture for highly scalable parallel machines are pointing toward large numbers of energy-efficient cores that are used as building blocks for large shared-memory nodes on a single chip or multichip module. Nodes such as those found on the IBM Blue Gene*/Q supercomputer are designed with upwards of 64 threads per node. In order to efficiently use such thread-rich computing environments, the parallel processing community has worked hard to unleash parallelism in applications of interest.

Today's applications targeting large parallel machines exhibit a wide range of parallelism constructs. Although there is no single parallel programming model, one model that is frequently used is as follows. At the highest level, coarse-grained parallelism is typically exploited using Message Passing Interface (MPI) [1] programs that explicitly communicate via messages. MPI processes are typically found across nodes, but nowadays multiple MPI processes also coexist within single shared-memory nodes. Medium-grained parallelism is often found in the outermost loops of a single MPI process and use shared memory within a node to communicate values between the phases of parallel computations. OpenMP** [2] is one of the most prevalent paradigms at this level, along with Unified Parallel C (UPC) [3] and explicitly threaded programs [4]. Fine-grain parallelism is typically found within the innermost loops of an application, with loops having from a few tens to a few hundreds of iterations. Nested parallelism with OpenMP is one of the predominant programming models at this level.

A significant fraction of OpenMP research is dedicated to efficiently encoding applications with OpenMP constructs and more recently with hybrid MPI/OpenMP constructs [5, 6]. Another significant effort aims at extending the programmability of OpenMP [7–9] and porting its programming model to platforms [9–14] other than uniform, shared-memory architectures. A third significant effort aims at better integrating traditional and OpenMP-specific compiler optimizations within the OpenMP framework [15–23]. However, we have found scant work on techniques to efficiently implement the basic OpenMP constructs such as the parallel and loop constructs that are prevalent in most OpenMP programs.

Initial experiments on the Blue Gene/Q indicate that the underlying algorithms used to implement the basic OpenMP constructs scale poorly [24–27] for the high numbers of threads. We have also found that the OpenMP overheads are typically too high to beneficially exploit the fine-grain parallelism present in applications. Low overheads are key when the amount of parallel work is small, because otherwise the overheads may quickly exceed the benefit of this limited form of parallelism.

In this paper, we present novel techniques to significantly lower the OpenMP overheads for the most prevalent

Digital Object Identifier: 10.1147/JRD.2012.2228769



**0018-8646/13/$5.00 © 2013 IBM**

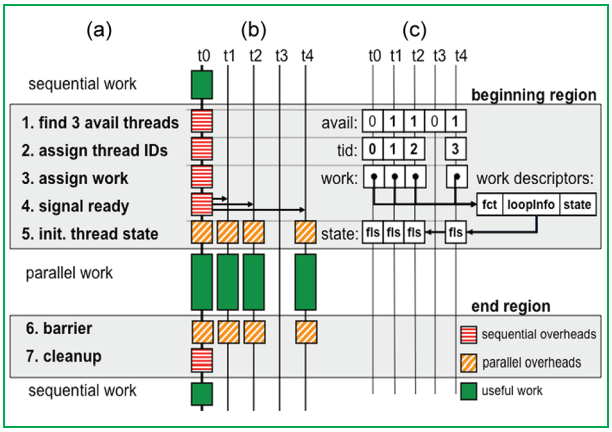



Figure 1

Anatomy of a parallel construct.

OpenMP construct. Here, we target the parallel constructs that are used to create parallel regions of code. By realizing that most applications repetitively allocate parallel regions of similar size, we can cache thread configurations to eliminate redundant computation and communication. When successful, we demonstrate near-constant overheads over a wide range of thread numbers. We present results for a Blue Gene/Q node using the ECPP Benchmark suites [24], a battery of tests designed for measuring OpenMP overheads. We reduce the parallel construct overheads by 1.9, 2.7, and 4.9 times when creating a team of 4, 16, and 64 threads, respectively.

The remainder of this paper is organized as follows. First, we present background on parallel constructs and describe our approach to improving their scalability. Then we detail our measurements and discuss related work.

## Parallel constructs

The parallel construct is the basic OpenMP primitives used to create parallel regions of code. There are multiple types, including constructs that jointly create a parallel region and distribute the iterations of a loop among the participating threads. We focus here on efficient implementation of the required interface for the basic construct, because it is the building block for the other parallel region constructs. We refer the reader to the OpenMP application programming interface (API) document [2] for details on the interface, such as methods to determine the numbers of threads to be assigned to a given parallel region or descriptions of which internal control variables must be copied to the private context of a thread.

### *Anatomy of a parallel construct*

Consider the typical flow of operations for the OpenMP parallel region depicted in **Figure 1**. At the highest level, thread $t0$ encounters a parallel region construct, depicted by the top light gray box. The runtime creates on its behalf a team of three additional threads, $t1$, $t2$, and $t4$ here. The four threads perform their parallel work, depicted with the series of four side-by-side dark green boxes. At the end of the parallel region, control is returned to the runtime to perform the OpenMP-required synchronization and cleanup operations, shown in the lower light gray box. Thread $t0$ then continues to execute further sequential work. Threads $t1$, $t2$, and $t4$ are returned to the pool of available OpenMP threads.

In OpenMP terminology, thread $t0$ is referred to as a *master thread*, because it creates a new team of threads. The other three threads are referred to as *worker threads*, because they assist the master thread in the parallel region. The master and the worker threads are referred to as a *team of threads* that jointly work on a given parallel region.

Figure 1(b) illustrates a timeline of the subtasks performed by the runtime during the beginning and end of a parallel region construct. Figure 1(c) describes the data structures touched during each subtask.

In Step 1 of Figure 1(a), the master thread identifies those threads that are available to participate in the parallel construct, for example, by using a compact bit-vector representation indicating which threads are available or busy. Selected threads are then marked as busy. This operation must be performed under a mutex because multiple master threads may run concurrently in the presence of nested parallelism.

In Step 2, the master thread assigns to each thread its own distinct thread number, referred to here as *thread identifier* (TID). Number 0 is reserved for the master thread, and successive integer numbers are assigned to the remaining worker threads. Here, the master thread writes the TIDs; subsequently, each thread in the team may access their TIDs, for example, to determine which thread is responsible to compute which subset of a given parallel loop.

In Step 3 of Figure 1(a), the runtime fills in a description of the work being requested by the user. Many computing environments encapsulate the parallel work in a compiler-generated function [15, 18, 19, 21, 28]; in such a case, the address of the function is inserted in the descriptor. Other information such as loop bounds (needed for combined parallel-for constructs) and ordered/nowait clauses can be entered there as well. Because each thread will later need to determine its workload, we must enter for each thread a pointer to the current work descriptor. We refer to such a pointer as a *work-descriptor identifier* (WID). Since both TID and WID are written by the master thread and are later read by each thread in the team, this data structure is typically implemented as a globally accessible array of (TID, WID) pairs, with one pair per thread in the system.

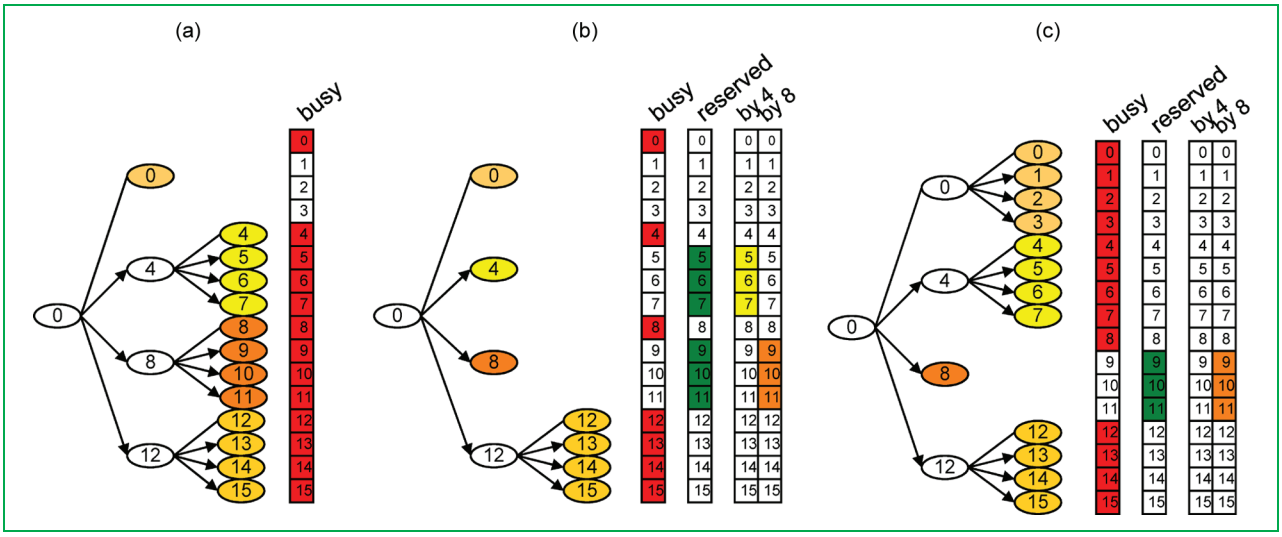



Figure 2

Caching the last set of worker threads of each master $t$, $TSET(t)$.

At this point, the worker threads have all of the required information necessary to start working. After being awaken in Step 4, they copy the global work description data in their own thread context. They are then ready to execute the user code in the parallel region.

After completing their parallel work, the threads return to the runtime and execute a synchronization barrier, as required by the OpenMP standard. The worker threads are placed back in the pool of available threads. Runtimes typically preserve idle worker threads because of the significant overhead associated with repetitive thread creation and destruction [15, 17]. The master thread performs further cleanup in Step 7, such as updating the global bit vector, indicating the available/busy status used in Step 1. It then returns to user control.

### *Overhead of a parallel construct*

To better understand the overhead of a parallel construct, we distinguish between two types of subtasks: the ones executed sequentially by the master and the ones executed in parallel by every thread in the team.

The most critical subtasks are the ones sequentially executed by the master threads. They are depicted in horizontally shaded red boxes in Figure 1(b). In addition, five of these six (Steps 1–4 and 7) have work that is proportional to the number of threads in the team ($TNUM$). For example, when assigning the TIDs in Step 2, the master thread must write a unique number for each thread in the team. Depending on the cache protocol, the communication cost may also be proportional to $TNUM$, as the master thread produces $TNUM$ data that is then consumed by $TNUM$, possibly remote worker threads.

The other subtasks are the ones executed in parallel by all of the threads in the team. They are shown in diagonally shaded orange boxes in Figure 1(b). The overheads associated with Step 5 mainly consist of communicating the work descriptor entry from the master to the worker's local cache. This step is clearly sensitive to the size of the work descriptor internal representation.

### *Near-constant-time thread allocation*

The key idea in lowering the overhead in Step 1 is to realize that most OpenMP applications repetitively request the same number of threads. We can significantly reduce the average overhead by caching the set of threads previously allocated in a parallel region with the expectation of reusing the same thread allocation for a future parallel region.

Consider the thread allocation shown in **Figure 2(a)**, depicting a nested parallel region of depth 2. The initial thread has created a first team including itself plus threads $t4$, $t8$, and $t12$. Defining here a *set of worker threads* for a given master $t$ as $TSET(t)$, we can state that $TSET(t0) = \{t4, t8, t12\}$. In turn, we see in Figure 2(a) that threads $t4$, $t8$, and $t12$ have become master threads of the teams with, respectively, $TSET(t4) = \{t5, t6, t7\}$, $TSET(t8) = \{t9, t10, t11\}$, and $TSET(t12) = \{t13, t14, t15\}$. The state of the available/busy bit vector is also shown in Figure 2(a), indicating with dark boxes the 13 busy threads $(t0, t4 - t15)$.

As stated above, the key idea is to preserve the $TSET$ information past the deallocation of a team. **Figure 2(b)** illustrates the state transition that occurs after the completion

of the parallel regions initiated by threads $t4$ and $t8$. Consider thread $t4$: as its team is deallocated, we preserve its $TSET(t4) = \{t5, t6, t7\}$ in the private execution context of $t4$. This state is depicted in Figure 2(b) as the third bit vector from the left. We also maintain a globally accessible "reserved" bit vector defined as the union of all thread sets currently preserved by any master threads in the system.

The maintaining of the global "busy" and "reserved" bit vectors allows the runtime to form new teams without clashing with both existing teams as well as with teams that have been created in the recent past.

Consider in **Figure 2(c)** the formation of a new four-thread team for thread $t0$. By checking the union of both the busy and the reserved bit vectors in Figure 2(b), the runtime can quickly determine that there are three remaining threads that are both available and not part of a prior team. By selecting these remaining threads ($t1$, $t2$, and $t3$), load balancing permitting, we maximize the chances of reusing past allocations.

Also shown in Figure 2(c), thread $t4$ can now allocate a four-thread team in constant time, by simply reusing the prior $TSET(t4)$ assignment saved in its private context. Thread $t4$ must simply ensure that its past allocation has not been invalidated, because this would have been the case if thread $t0$ had requested a larger team.

To ensure correctness, we introduce a globally accessible generational counter. Whenever a master thread preserves an allocation, it also saves in its context the current value of the global generational counter. In any subsequent thread allocations, if a master thread needs to use threads that are marked as reserved, this master thread also increments the global generational counter, thereby invalidating every saved allocation. When a master thread attempts to create a new team of size no larger than its saved configuration, it simply checks that both the local and global values of the generational counter match. When they match, the saved allocation is valid; when they do not, the saved allocation may not be valid and is thus discarded. Although other policies could be used, this simple policy works well in practice and has constant-time overheads.

To summarize the finding in this section, we can determine in constant time whether a past thread allocation is valid or not by using a generational counter. If valid, the allocation can be finalized with a couple of bit-vector operations. Current architectures allow constant-time bit-vector operations of up to 128 to 256 bits, depending on the SIMD (single-instruction, multiple-data) width of the host processor.

### Eliminating assignment notification overheads

The main idea in removing the overhead associated with the per-thread assignment of a unique TID and WID is to piggyback on the successful reuse of prior thread allocations described in the subsection "Near Constant-Time Thread-Allocation."

Recall that in Steps 2 and 3 in Figure 1(a), the master thread writes information in the global (TID, WID) array that allows each thread in the team to uniquely determine its assigned work. When successfully reusing a prior thread allocation, we can reuse the same TID mapping because by definition of our caching policy, the current request is performed by the same master and the current request asks for a team of size no larger than the size of the cached team allocation.

The next step in eliminating overheads is to ensure that we do not need to modify the WIDs of the threads participating in the team. To do so, we also set aside the work-descriptor entry associated with a parallel region when freeing an allocation and preserving its thread allocation. When successfully reusing a prior thread allocation, we then reuse the work descriptor entry set aside, thus bypassing any need to update the assignment table. As a result, the TID and WID entries in the assignment table remain local to the cache of each of the worker threads because they were not modified by the master thread. Thus, we avoid redundant computation as well as redundant communication. Only the content of the work-descriptor entry itself is modified.

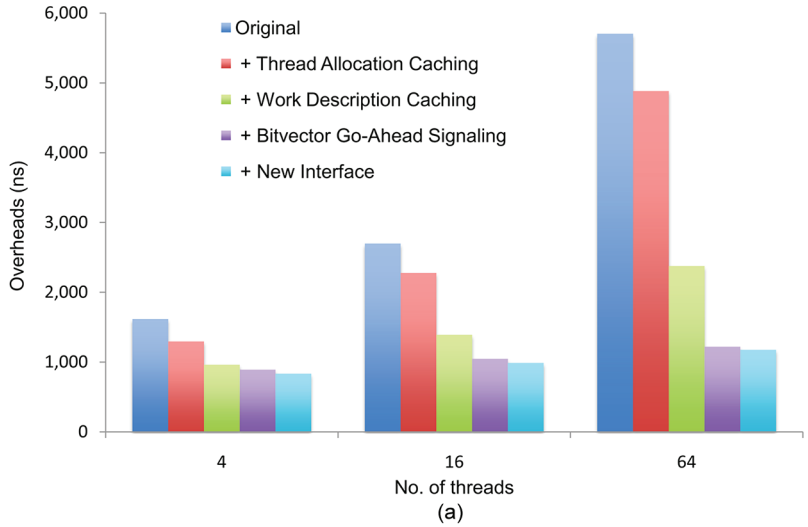
### Reducing the "go-ahead" signaling overheads

Runtimes built on top of a *pthread* library [15] may signal the worker threads to exit their idling loop [Step 4 in Figure 1(a)] by using pairs of mutex and condition [4]. Other runtimes may use a per-thread variable whose content is changed by the master and whose change is detected by the idling worker.

We use here a single globally accessible bit vector, subsequently referred as the "activate" bit vector. To activate a set of thread $TSET(t)$, the master simply performs an atomic-XOR of the current $TSET(t)$ with the activate bit vector. Each idling thread recalls the last value of its bit in the activate bit vector. When an idling thread notices that its bit has been flipped, it then knows that there is new work to be fetched. Work is found by accessing the (TID, WID) pair assigned to it in the global assignment table.

### Improved application-runtime interface

Modifying the interface between the application program and the OpenMP runtime can also result in lower overhead. For example, the runtime often needs to retrieve a pointer to internal runtime data that is private to each thread. Localizing such thread-private pointers can be expensive. This search can often be avoided by first providing this pointer to the application and then having the application pass this pointer back when performing a subsequent call to the runtime. Another example is that many calls to the runtime mostly use default values for a long list of possible switches (refer to clauses in OpenMP). By providing



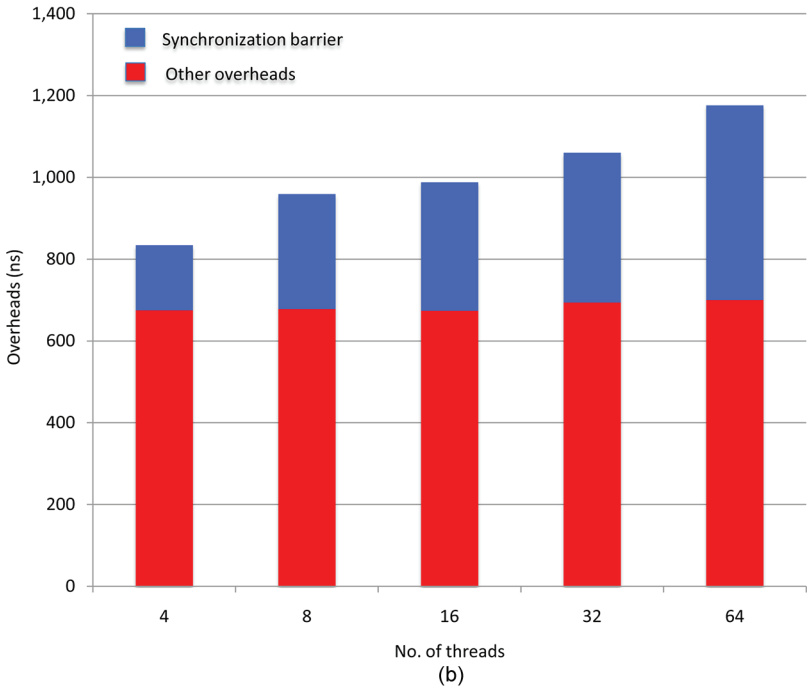




Figure 3

(a) Reduction in overhead due to proposed techniques. (b) Near-constant non-barrier-related overheads.

two entry points, one accepting all legal switches and one assuming default values, this special case allows the latter entry point to execute faster because many special cases in the runtime can be statically eliminated.

## Measurements

In this paper, we use the Edinburgh Parallel Computing Center (EPCC) benchmark suite [24] that was developed to specifically measure the OpenMP overheads caused by synchronization and loop scheduling. The underlying technique is to compare the time taken for a section of code executing sequentially one unit of work to the time taken for the same code executing in parallel *TNUM* units of work. All comparisons of runtime overheads are performed in the same runtime code base and compiled with the same options (−O3 with inter-procedural analysis and aggressive

in-lining). It should be noted that the experimental runtime used here provides no support for OpenMP tasks; we estimate that adding such support should not have an impact on the results presented here by more than 10%. All measurements are performed on a single Blue Gene/Q node. We always fully populate the hardware threads of each core: 4-thread results use 1 core, 16-thread results use 4 cores, and 64-thread results use 16 cores.

**Figure 3(a)** illustrates the benefits of turning on each of the optimizations described in the section "Parallel constructs." For each set of five bars, the first bars correspond to a traditional implementation of the runtime, as described in the section "Overhead of a Parallel Construct." Creating a parallel region takes 1.6 $\mu$s, 2.7 $\mu$s, and 5.7 $\mu$s for teams of 4, 16, and 64 threads, respectively. Clearly, overheads are very dependent on the size of the team.

The second bar in each set illustrates the benefits of reusing past thread allocations, as proposed in the section "Near-constant-time thread allocation." The benefits should be higher for larger team sizes, because successful caching bypasses a linear-cost search for available threads. Indeed, Figure 3(a) indicates a reduction in overhead of 300 ns, 400 ns, and 800 ns for teams of size 4, 16, and 64 threads, respectively.

The third bar in each set corresponds to also reusing the assignment notifications when successfully reusing thread allocation, as described in the section "Eliminating assignment notification overheads." The combined effect of these two optimizations cuts the parallel construct overheads by a factor of 1.7, 1.9, and 2.4 for teams of 4, 16, and 64 threads, respectively, compared with the original overhead.

The fourth bar in each set corresponds to adding to the previous runtime the fast go-ahead notification using a single XOR operation, as proposed in the section "Reducing the "go-ahead" signaling overheads." This again removes a linear component to the overheads, as demonstrated by the larger reductions in overhead as larger teams are created. The last bar in each set corresponds to adding the various interface optimizations described in the section "Improved application-runtime interface." This component only lowers the overhead by a few percentage points (more so for larger teams).

Combined, the improvements proposed in the section "Parallel constructs" reduce the overheads depicted in Figure 3(a) by 1.9, 2.7, and 4.9 times, compared with the original overheads for team sizes of 4, 16, ands 64 threads, respectively.

Because the parallel constructs include by definition a barrier at the end of the construct, and because the cost of barrier synchronization is proportional to the number of threads to synchronize, we cannot easily tell from Figure 3(a) how the other OpenMP overheads scale. In **Figure 3(b)**, we have separated the synchronization costs (blue bars) from the remaining parallel construct costs (red bars) for our last, most optimized scheme. The other OpenMP overheads range from 675 to 700 ns across the range of 4 to 64 threads. These overheads include acquiring two locks to update the thread allocation table plus handling all of the OpenMP internal control variables as mandated by the standard. For comparison, these same overheads range from 1,400 to 5,180 ns in the original configurations. In other words, the non-synchronization OpenMP overheads of a 64-thread parallel region are less than half of what they used to be for a 4-thread parallel region.

## Related work

We describe here work related to the implementation and optimization of the OpenMP programming environment with respect to the handling of parallel loop and parallel region constructs.

Transforming parallel regions into compiler-generated outline functions [29] was proposed prior to OpenMP and has been used since by most OpenMP runtimes [15, 18, 19, 21, 28, 30]. Some runtimes have outlined functions in the scope of the original parallel region [19, 21] to reduce overheads for shared variables. Recent work has embedded the work directly in the original function using multi-entry threading constructs [16] to facilitate advanced optimizations.

In addition to transforming OpenMP directives, compilers have optimized parallel regions early on [29, 30]. The OpenUH portable compiler [19] has provided both a platform-independent source-to-source optimizing translator as well as a platform-dependent integrated compiler for supporting optimizations such as loop transformations for cache locality. Global optimizations were later added to their framework using parallel control-flow graphs [22] as well as cost models to better direct the compiler [20]. The ORC-OpenMP compiler classifies OpenMP directives with the aim of simplifying or eliminating calls to the runtime, for example, for known sequential OpenMP regions. Intel's support for OpenMP [16, 17] includes a highly optimized compiler that performs, among others, in-lining, aggressive code motion, and redundancy elimination over OpenMP constructs internally represented as multi-entry threading regions. The ROSE compiler [31] aims at providing portable OpenMP performance across multiple platforms and associated runtime environment. Others have suggested tools for automatic characterizations of application memory sharing patterns to tune OpenMP constructs [32].

The performance of barriers has been studied in the context of OpenMP [33]. Compiler optimizations to simplify or eliminate OpenMP barriers have also been proposed [23]. Significant work in OpenMP has focused on applying the OpenMP shared-memory paradigm to other architectures, including a network of workstations [10], clusters of possibly shared-memory machines [11, 12], NUMA machines [9], and more recently, gaming platforms such as the Cell Broadband Engine** [13] and general-purpose graphics

processing units [14]. Others have proposed important extensions to the OpenMP interface, along with related OpenMP optimizations and runtimes, for example, for better locality handling [7, 9] and finer control of nested parallelism [8].

## Conclusions

As the number of threads found in parallel supercomputers such as the Blue Gene/Q continues to increase, there is a growing need on applications to exploit more parallelism in their code, including coarse-, medium-, and fine-grain parallelism.

OpenMP is one of the dominant shared-memory programming models and is well suited for exploiting medium- and fine-grain parallelism. Although most OpenMP research has focused on application tuning, integration within optimizing compilers, programming model extensions, and porting to distributed-memory platforms, we have found that current algorithms used to implement basic OpenMP constructs have large overheads and scale poorly.

In this paper, we have addressed these scaling issues for creating parallel regions. By exploiting thread-allocation reuse, we have demonstrated reductions in overheads by up to 5 times when creating parallel teams of up to 64 threads. The proposed algorithm scales very well, as its non-synchronization overheads are nearly constant when creating parallel regions over a wide range of thread team size. Now that the OpenMP overheads have been significantly lowered, it should be possible for more applications to take advantage of the fine-grain parallelism present in their code.

## Acknowledgments

The IBM Blue Gene/Q project has been supported and partially funded by Argonne National Laboratory and the Lawrence Livermore National Laboratory on behalf of the U.S. Department of Energy, under Lawrence Livermore National Laboratory subcontract no. B554331. In addition, we thank Kit Barton, Tao Liu, Raul Silvera, Priya Unnikrishnan, and Amy Wang for sharing their expertise on OpenMP runtimes.



## References

1. M. Snir, S. Otto, S. Huss-Lederman, and D. Walker, *MPI-The Complete Reference*. Cambridge, MA: MIT Press, 1998.
2. OpenMP Application Programing Interface, May 2008, Version 3.0. [Online]. Available: http://openmp.org
3. W. W. Carlson, J. M. Draper, and D. E. Culler, "Introduction to UPC and language specification," Univ. California-Berkeley, Berkeley, CA, TR CCS-TR-99-157, 1999.
4. D. R. Butenhof, *Programming With Posix Thread*. Reading, MA: Addison-Wesley, 1997.
5. F. Cappello and D. Etiemble, "MPI versus MPI+OpenMP on the IBM SP for the NAS benchmarks," in *Proc. Conf. Supercomput.*, 2000, p. 12.
6. L. Smith and M. Bull, "Development of mixed mode MPI/OpenMP applications," *J. Sci. Program.*, vol. 9, no. 2/3, pp. 83–98, Aug. 2001.
7. B. Chapman, P. Mehrotra, and H. Zima, "Enhancing OpenMP with features for locality control," in *Proc. 8th ECMWF Workshop Parallel Process. Meteorol.—Towards Teracomput.*, 1998, pp. 301–313.
8. E. Ayguade, M. Gonzalez, J. Labarta, X. Martorell, N. Navarro, and J. Oliver, "NanosCompiler: A research platform for OpenMP extensions," in *Proc. Eur. Workshop OpenMP*, 1999, pp. 27–31.
9. J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C. A. Nelson, and C. D. Offner, "Extending OpenMP for NUMA machines," in *Proc. ACM/IEEE Supercomput.*, Washington, DC, 2000, (CDROM).
10. H. Lu, Y. C. Hu, and W. Zwaenepoel, "OpenMP on networks of workstations," in *Proc. Conf. Supercomput.*, 1998, pp. 1–15.
11. M. Sato, S. Satoh, K. Kusano, and Y. Tanaka, "Design of OpenMP compiler for an SMP cluster," in *Proc. Eur. Workshop OpenMP*, 1999, pp. 32–39.
12. L. Huang, B. Chapman, and R. Kendall, "OpenMP for clusters," in *Proc. Eur. Workshop OpenMP*, 2003, pp. 22–26.
13. K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on cell," *Int. J. Parallel Program.*, vol. 36, no. 3, pp. 289–311, Jun. 2008.
14. S. Lee, S.-J. Min, and R. Eigenmann, "OpenMP to GPGPU: A compiler framework for automatic translation and optimization," in *Proc. Symp. Princ. Pract. Parallel Program.*, 2009, pp. 101–110.
15. C. Brunschen and M. Brorsson. (2000, Oct.). OdinMP/CCp—A portable implementation of OpenMP for C. *Concurrency, Pract. Exp.* [Online]. *12(12)*, pp. 1193–1203. Available: http://parallel.ksu.ru/ftp/openmp/ewomp99.pdf
16. X. Tian, A. Bik, M. Girkar, P. Grey, H. Saito, and E. Su, "Intel OpenMP C++/Fortran compiler for hyper-threading technology: Implementation and performance," *Intel Technol. J.*, vol. 6, no. 1, pp. 1–11, 2002.
17. X. Tian, M. Girkar, S. Shah, D. Armstrong, E. Su, and P. Petersen, "Compiler and runtime dupport for running OpenMP programs on pentium- and itanium-architectures," in *Proc. Int. Workshop High-Level Paralllel Program. Models Support. Environ.*, 2003, pp. 47–55.
18. Y. Chen, J. Li, S. Wang, and D. Wang, "ORC-OpenMP: An OpenMP compiler based on ORC," in *Proc. Int. Conf. Comput. Sci.*, 2004, vol. 3038, pp. 414–423, Lecture Notes in Computer Science.
19. C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng, "OpenUH: An optimizing, portable OpenMP compiler," *Concurrency Comput., Pract. Exp.*, vol. 19, no. 18, pp. 2317–2332, Dec. 2007.
20. C. Liao and B. Chapman, "Invited paper: A Compile-time cost model for OpenMP," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, 2007, pp. 1–8.
21. X. Teruel, P. Unnikrishnan, X. Martorell, E. Ayguade, R. Silvera, G. Zhang, and E. Tiotto, "OpenMP tasks in IBM XL compilers," in *Proc. Conf. Center Adv. Stud. Collab. Res.*, 2008, pp. 16:207–16:221.
22. L. Huang, D. Eachempati, M. W. Hervey, and B. Chapman, "Exploiting global optimizations for OpenMP programs in the OpenUH compiler," in *Proc. Symp. Princ. Pract. Parallel Program.*, 2009, pp. 289–290.
23. H. Ma, R. Zhao, X. Gao, and Y. Zhang, "Barrier optimization for OpenMP program," in *Proc. Int. Conf. Softw. Eng., Artif. Intell., Netw. Parallel/ Distrib. Comput.*, 2009, pp. 495–500.
24. J. M. Bull, "Measuring synchronisation and scheduling overheads in OpenMP," in *Proc. Eur. Workshop OpenMP*, 1999, pp. 99–105.

25. C. Liao, Z. Liu, L. Huang, and B. Chapman, "Evaluating OpenMP on chip multithreading platforms," in *Proc. 1st Int. Workshop OpenMP*, 2005, pp. 178–190.
26. G. Bronevetsky, J. Gyllenhaal, and B. R. de Supinski, "CLOMP: Accurately characterizing OpenMP application overheads," *Int. J. Parallel Program.*, vol. 37, no. 3, pp. 250–265, Jun. 2009.
27. K. Fürlinger and M. Gerndt, "Analyzing overheads and scalability characteristics of OpenMP applications," in *Proc. 7th Int. Meeting High Perform. Comput. Comput. Sci.*, 2006, pp. 39–51.
28. *The GNU OpenMP Implementation,* Free Softw. Found., Boston, MA, 2011. [Online]. Available: http://gcc.gnu.org/onlinedocs/libgomp.pdf
29. J.-H. Chow, L. E. Lyon, and V. Sarkar, "Automatic parallelization for symmetric shared-memory multiprocessors," in *Proc. Conf. Centre Adv. Stud. Collab. Res.*, 1996, p. 5.
30. G. Zhang, R. Silvera, and R. Archambault, "Structure and algorithm for implementing OpenMP workshares," in *Proc. 5th Int. Conf. OpenMP Appl. Tools—Shared Memory Parallel Program. OpenMP*, 2005, vol. 3349, pp. 110–120, Lecture Notes in Computer Science.
31. C. Liao, D. J. Quinlan, T. Panas, and B. de Supinski, "A ROSE-based OpenMP 3.0 research compiler supporting multiple runtime libraries," in *Proc. Int. Workshop OpenMP*, 2010, pp. 15–28.
32. J. Marathe and F. Mueller, "Source-code-correlated cache coherence characterization of OpenMP benchmarks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 6, pp. 818–834, Jun. 2007.
33. R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin, "Scalability evaluation of barrier algorithms for OpenMP," in *Proc. 5th Int. Workshop OpenMP*, 2009, pp. 42–52.

*Received March 16, 2012; accepted for publication July 26, 2012*

**Alexandre E. Eichenberger** *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (alexe@us.ibm.com).* Dr. Eichenberger is a Research Staff Member in the Advanced Compiler Technologies Department at the IBM T. J. Watson Research Center. He received a diploma in computer engineering from the Eidgenössische Technische Hochschule, Zürich, in 1991, and a Ph.D. degree from the University of Michigan, Ann Arbor, in 1996. He joined IBM in 2001, with a research focus on the interaction of compiler technology and microarchitecture design. Interests include efficient support for parallelism on upcoming exascale supercomputers, compiler technology for parallelization and SIMD code generation, and efficient scheduling algorithms. In 1997, he received an IBM Outstanding Technical Achievement Award for contributions to the Cell Broadband Engine microprocessor. He is author or coauthor of more than 25 technical papers and 12 patents and is an IBM Master Inventor.

**Kevin O'Brien** *IBM Research Division, Thomas J. Watson Research Center, Yorktown Heights, NY 10598 USA (caomhin@us.ibm.com).* Mr. O'Brien holds a B.Sc. degree in theoretical physics and an M.Sc. degree in astrophysics from Queen Mary College, London University, England. He is currently manager of Advanced Compiler Technology at the IBM T. J. Watson Research Center. He has focused his efforts on deploying diverse compiler optimization skills to the improvement of IBM products from XLC, XLF, XLC (C++), and Smalltalk compilers to the design of a dynamic optimizer for a just-in-time compiler for a Java** virtual machine (JVM**) for the first Transmeta machine. More recently, he has been involved in research in advanced parallelization techniques based on polyhedral theory. The results of this research have been incorporated in the IBM XL product compilers. He has also contributed to the architectures of POWER*, PowerPC*, and Cell Broadband Engine. He has received an IBM Corporate Award and Outstanding Technical Achievement Awards for his work on the XL compilers. He holds 20 U.S. patents including one on dynamic optimization and one on speculative multithreading.