

Scalability evaluation of barrier algorithms for OpenMP

Ramachandra Nanjegowda, Oscar Hernandez,
Barbara Chapman

Computer Science Department

University of Houston, Houston, Texas 77004, USA
rchakena@uh.edu, oscar@cs.uh.edu, chapman@cs.uh.edu

Haoqiang H. Jin

NASA Ames Research Center, Moffett Field, CA 94035
haoqiang.jin@nasa.gov

April 24, 2009

Abstract

OpenMP relies heavily on barrier synchronization to coordinate the work of threads that are performing the computations in a parallel region. A good implementation of barriers is thus an important part of any implementation of this API. As the number of cores in shared and distributed shared memory machines continues to grow, the quality of the barrier implementation is critical for application scalability. There are a number of known algorithms for providing barriers in software. In this paper, we consider some of the most widely used approaches for implementing barriers on large-scale shared-memory multiprocessor systems: a “blocking” implementation that de-schedules a waiting thread, a “centralized” busy wait and three forms of distributed “busy” wait implementations are discussed. We have implemented the barrier algorithms in the runtime library associated with a research compiler, OpenUH. We first compare the impact of these algorithms on the overheads incurred for OpenMP constructs that involve a barrier, possibly implicitly. We then show how the different barrier implementations influence the performance of two different OpenMP application codes.

1 Introduction

OpenMP programs typically make repeated use of barriers to synchronize the threads that share the work they contain. Implicit barriers are required at the end of parallel regions; they are also required at the end of worksharing constructs unless explicitly suppressed via a *NOWAIT* clause. Explicit barriers may

be inserted elsewhere by the application developer as needed in order to ensure the correct ordering of operations performed by concurrently executing, independent threads. The OpenMP implementation will therefore need to include barrier synchronization operations: these are typically an important part of the runtime library, whose routines are inserted into OpenMP code during compilation and subsequently invoked during execution. Given the importance of this construct in the OpenMP API, a good barrier implementation is essential. In OpenMP 3.0, threads waiting at barriers are permitted to execute other tasks. Hence eliminating barrier contentions is essential to free up threads or improve work stealing. As the number of threads in parallel regions steadily grows, and the memory hierarchies in the cores and parallel machines become more complex, the scalability of the implementation becomes increasingly important. OpenMP implementations should provide a choice of barrier implementations for different environments and architectures with complex memory hierarchies and interconnect topologies.

Barrier algorithms are generally considered to fall into two classes: those based upon a blocking construct that de-schedules a waiting thread, and those relying on a busy-wait construct, in which threads loop on a shared variable that needs to be set or cleared before they can proceed. The primary disadvantage of scheduler-based blocking is that the overhead of scheduling may exceed the expected wait time. The scheduling overhead we are referring here is the overhead involved in switching the thread back and forth when it is blocked. The operating system scheduler will switch the blocked thread and relinquish the processor, for any other thread which is ready to execute. On the other hand, the typical implementation of busy-waiting introduces large amounts of memory and interconnect contention which causes performance bottlenecks. Hence enhanced versions of a busy-wait barrier implementation, generally known as distributed busy-wait, have been devised. The key to these algorithms is that every processor spins on a separate locally-accessible flag.

As part of an effort to investigate ways in which OpenMP and its implementations may scale to large thread counts [9], we have begun to study a variety of strategies for accomplishing the most expensive synchronization operations implied by this API, including barriers and reductions. We believe that the existing set of features for expression of the concurrency and synchronization within an application must be enhanced in order to support higher levels of shared memory concurrency, but that a careful implementation of existing features may go some of the way to improving the usefulness of this programming model on large shared-memory machines. The aim of the work described here is to gain insight into the behavior of different barrier algorithms in the OpenMP context in order to determine which of them is most appropriate for a given scenario. Our overall goal is to provide an OpenMP library which will adapt to deliver the most suitable implementation of a barrier based on the number of threads used, the architecture (memory layout/interconnect topology), application and possibly system load.

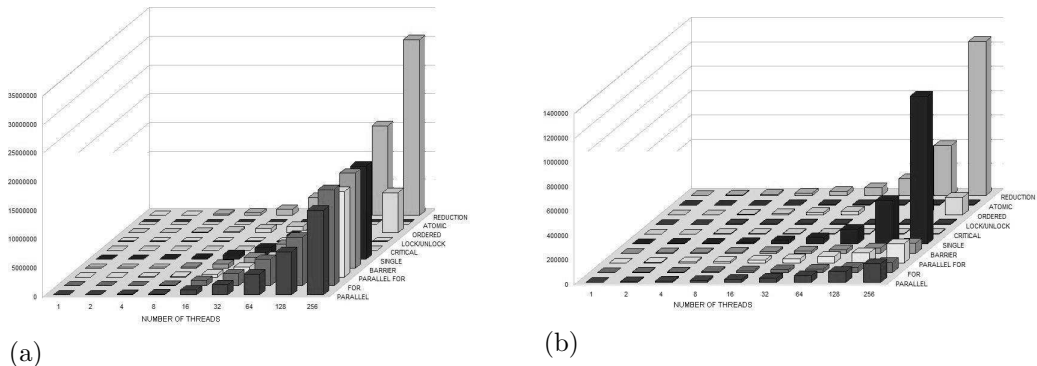


Figure 1: (a) The performance of EPCC syncbench compiled using OpenUH. (b) The performance of EPCC syncbench compiled using Intel compiler.

2 OpenUH Implementation of OpenMP

The experimental results described here are based upon barrier algorithms implemented in the runtime library associated with the OpenUH compiler. OpenUH [1], a branch of the Open64 [2] compiler suite, is an optimizing and portable open-source OpenMP compiler for C/C++ and Fortran 95 programs. The suite is based on SGI's Pro64 compiler which has been provided to the community as open source. At the time of writing, OpenMP 2.5 is supported along with a partial implementation of OpenMP 3.0; the compiler's OpenMP runtime library is open source. It targets the Itanium, Opteron, and x86 platforms, for which object code is produced, and may be used as a source-to-source compiler for other architectures using the intermediate representation (IR)-to-source tools.

OpenUH translates OpenMP to an internal runtime API, which provides the internal data structures and thread management needed to implement OpenMP constructs. The OpenMP runtime uses the Posix Threads API for thread creation, thread signals, and locks with the goal of providing a portable OpenMP runtime implementation. The initial barrier implementation in OpenUH was based on a very straightforward centralized blocking barrier algorithm, described in the next section, and which was known to have poor scalability.

In Figure 1, we show the overheads of the different OpenMP constructs as the number of processors increases on a SGI Altix 3700 using the EPCC microbenchmarks [7]. The graph on the left gives the overheads in milliseconds reported for OpenUH, and those of the Intel 10.1 compiler are given on the right. A quick inspection of these results shows that there are OpenMP operations which are significantly more expensive than others in both implementations: these include reductions, single constructs, parallel/parallel for and barriers. It is important to note that the overhead of such constructs depends on the way the barrier is implemented, since they are either the barrier or they contain an implicit barrier.

3 Approaches to Implementing A Barrier

In this section we describe several different implementations of the barrier construct that we considered for this work. All of them appear in the literature, and have been implemented in different parallel languages and libraries, including MPI, UPC, CoArray Fortran and Global Arrays. Our interest was to evaluate them with regard to their usefulness for OpenMP and to determine whether one or more of them provided superior performance. We use a pseudo code representation to describe the algorithms.

3.1 Centralized Blocking Barrier

This barrier implementation is based on a single thread counter, a mutex and a condition variable. The counter keeps track of the number of threads that have reached a barrier. If the counter is less than the total number of threads, the threads do a condition wait. The last thread to reach this barrier wakes up all the threads using condition broadcast. The mutex is used to synchronize access by the different threads to the shared counter.

The need to acquire the mutex lock is the main bottleneck in this algorithm, as all the threads compete to acquire it when they receive the broadcast signal. An alternative implementation uses multiple condition variables and mutexes to relieve this. A pair of threads will then use a particular set of mutex and condition variables, which will provide somewhat better performance than is obtained by using a single mutex and condition variable.

Under both these strategies, the scheduling overhead (the overhead of context switching the blocked thread) is far more expensive than the typically expected barrier wait time, and hence this simplistic algorithm is not an efficient way to implement a barrier where thread blocking is not needed. However this algorithm makes sense when we oversubscribe threads to cores (i.e. several threads share a core), because it frees up CPU resources allowing other threads to obtain CPU time.

3.2 Centralized Barrier

In a centralized barrier algorithm, each processor updates a counter to indicate that it has arrived at the barrier and then repeatedly polls a flag that is set when all threads have reached the barrier. Once all threads have arrived, each of them is allowed to continue past the barrier. The flag can be a sense reversal flag, to prevent intervention of adjacent barrier operations.

This implementation uses a small amount of memory, is simple to implement, and could be good if cores share small fast caches (typically L2).

The potential drawback of centralized barriers is that the busy waiting to test the value of the flag occurs on a single, shared location. As a result, centralized barrier algorithms cannot be expected to scale well to large numbers of threads. Our experiments (see Section 4 below) confirm this. However, because of the

```

// Shared data.
barrier_lock // pthread mutex lock
barrier_cond // pthread condition variable
barrier_count // Number of threads not yet arrived
barrier_flag // To indicate all arrived

procedure blocking_barrier
  barrier_flag=team->barrier-flag
  new_count = atomic_incr(barrier_count)

  if(new_count==team_size)
    team->barrier_count=0
    team->barrier_flag = barrier_flag ^ 1
    pthread_cv_broadcast()
  else
    pthread_mutex_lock()
    while(barrier_flag==team->barrier_flag)
      pthread_cv_wait()
    pthread_mutex_unlock()

```

(a) Blocking Barrier Algorithm

```

// Shared data.
barrier_count // not yet arrived
barrier_flag // indicate all have arrived
team_size // number of threads

procedure central_barrier
  barrier_flag=team->barrier-flag
  new_count = atomic_inc(barrier_count)

  if (new_count == team_size)
    team->barrier_count = 0
    team->barrier_flag = barrier_flag ^ 1
  else
    while (team->barrier_flag == barrier_flag)

```

(b) Centralized Barrier Algorithm

small amount of memory used, it may be a contender where cores share cache lines, as it keeps the rest of the cache almost intact.

3.3 Dissemination Barrier

```

// Shared data:
int P // number of threads
struct node {
  boolean flag[2]
  struct node *partner
}
node nodes[P][logP] // array of nodes

// Private data for each thread:
volatile int parity
volatile int sense

// initializes each thread
// to its partner
procedure dissem_init() {
  for (i = 0; i < P; i++) {
    d = 1;
    for (r = 0; r < logP; r++) {
      nodes[i][j].partner=nodes[(i+d) % P][j];
      d = 2*d;
    }
  }
}

```

(a) Dissemination Barrier Initialization

```

procedure dissem_barrier {
  i = thread_id;
  sense = thread_private->sense
  parity = thread_private->parity
  for ( r = 0; r < logP; r++) {
    nodes[i][r].partner.flag[parity]= sense;
    while (nodes[i][r].flag[parity] != sense)
  }

  if(parity==1)
    thread_private->sense = sense^1;
    thread_private->parity=1-parity;
}

```

(b) Dissemination Barrier Algorithm

This barrier implementation technique derives its name from the way in which it disseminates information among a set of threads. Each thread spins around a variable dedicated to it, and signaled by another thread. Each thread goes through $\log(N)$ rounds where N is the number of threads. At the end of

rounds, the thread knows that the other threads in the system have reached the barrier and it is good to proceed to the next barrier episode.

In step k , thread i signals thread $(i+2k) \bmod P$. For each step, we use alternate sets of variables to prevent interference in consecutive barrier operations. This algorithm also uses sense reversal to avoid resetting variables after every barrier. The dissemination barrier has shorter critical as compared to other algorithms. Our experiments (see Table 1) shows that this algorithm gives the best performance upto 16 threads. It also gives best performance for the barrier construct on multicore platform (see Table 2).

3.4 Tree Barrier

In this method, each thread is assigned to a unique tree node which is linked into an arrival tree by a parent link and into the wakeup tree by a set of child links. The parent notifies each of its children by setting a flag in the nodes corresponding to them. The child in turn sets a flag in the parent node to signal its arrival at the barrier.

The data structures for the tree barrier is initialized such that each node's parent flag variable points to the appropriate *childnotready* flag. The *child_notify* variable points to the *wakeup_sense* variable. The *havechild* flag indicates whether a particular node has children or not. During a barrier phase, a thread tests to see if the *childnotready* flag is clear for each of its children before reinitializing them to next barrier. After all children of a node have arrived, the *childnotready* flag is cleared. All threads other than root spins on their local *wakeup_sense* flag. At each level, a thread releases all its children before leaving the barrier and thus eventually the barrier is complete.

3.5 Tournament Barrier

The Tournament barrier algorithm is similar to a tournament game. Two threads play against each other in each round. The loser thread sets the flag on which the the winner is busy waiting. Then the loser thread waits for the global champion flag to be set, where as the winners, play against each other in next round. The overall winner becomes the champion and notifies all losers about the end of barrier.

The complete tournament barrier requires $\log N$ "rounds", where N is the number of threads. The threads begin at the leaves of the binary tree and at each step, one thread continues up to the tree to the next round of the tournament. The WINNER and LOSER at each stage is statically determined during initialization. In round k , thread i sets a flag of thread j , where $i = 2^k$, and $j = (i - 2)^k$. The LOSER thread i drops out and busy waits on a global flag while the WINNER thread j , participates in the next round of the tournament.

```

// Shared data:
typedef struct {
    volatile boolean *parentflag;
    boolean *child_notify[2];
    volatile long havechild;
    volatile long childnotready;
    boolean wakeup_sense;
} treenode;
treenode shared_array[P];

Private data for each thread:
volatile int parity;
volatile boolean sense;
treenode *mynode;

procedure tree_barrier
vpid=thread_id;
treenode *mynode_reg;
mynode_reg = cur_thread->mynode;

while (mynode_reg->childnotready);

mynode_reg->childnotready = mynode_reg->havechild;
*mynode_reg->parentflag = False;

if (vpid)
    while(mynode_reg->wakeup_sense != cur_thread->sense);

*mynode_reg->child_notify[0] = cur_thread->sense;
*mynode_reg->child_notify[1] = cur_thread->sense;
cur_thread->sense ^= True;
}

// Shared data:
struct round_t
{
    boolean *opponent
    int role // WINNER or LOSER
    boolean flag
}
round_t rounds[P][logP]

// Private data for each thread:
boolean sense
int parity
round_t *myrounds

procedure tour_barrier
round_t round=current_v_thread->myrounds;
for(;;) {
    if(round->role & LOSER) {
        round->opponent->flag = sense;
        while (root_sense != sense);
        break;
    }
    else if (round->role & WINNER)
        while (round->flag != sense);
    else if (round->role & ROOT) {
        while (round->flag != sense);
        champion_sense = sense;
        break;
    }
    round++;
}

```

(a) Tree Barrier Algorithm

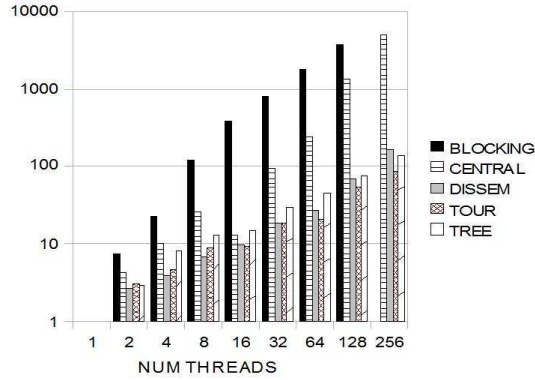
(b) Tournament Barrier Algorithm

4 Performance Measurements

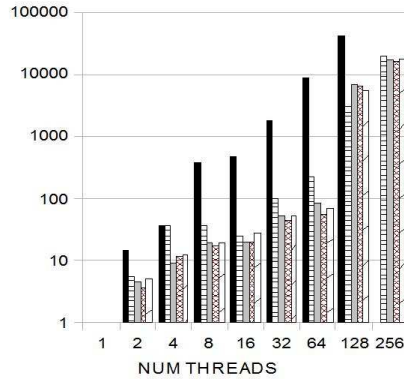
We have tested the barrier implementations described above on several different platforms. Experiments were performed on Cobalt, NCSA's SGI 3700 Altix, consisting of two systems each with 512 Intel 1.6 Ghz Itanium2 Madison processors running SUSE 10.2 (see <http://www.teragrid.org/> for a detailed description of the system). The experiments up to 512 threads was run on Columbia, NASA's SGI 4700 Altix, consisting of 1024 dual-core Intel 1.6 Ghz Itanium2 Montecito processors running SUSE Linux Enterprise Operating system (see <http://www.nas.nasa.gov/> for a detailed description of the system). Other experiments were conducted on a Sun Fire X4600 with eight dual core AMD Opteron 885 processors and a Fujitsu-Siemens RX600S4/X system with four Intel Xeon E7350 quad core processors located a Aachen University.

4.1 EPCC Microbenchmark Results

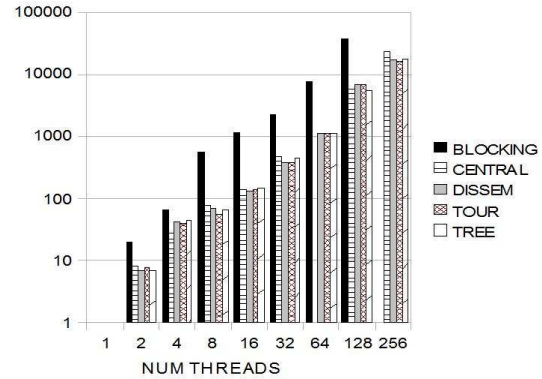
We implemented each of the five algorithms described above in the OpenUH runtime library and used the corresponding portion of the EPCC microbenchmarks to test the barrier performance they supply. The first diagram (See Fig a) gives the time taken to implement a barrier in microseconds for 2, 4, 16, 32, 64,



(a) BARRIER



(b) PARALLEL



(c) REDUCTION

128 and 256 threads on SGI 3700. As can be clearly seen, the centralized blocking and centralized barrier algorithm did not perform as well as the tournament and dissemination barrier. This is expected since our test cases didn't involve oversubscribing threads to cores. The tournament algorithm resulted in the least overhead where the thread count is greater than 16 and the dissemination barrier is best when the number of threads is less than 16. For space reasons, we did not show results of our tests on a Sun Fire X4600 and a Fujitsu-Siemens Intel Xeon with up to 16 threads. In both systems, the dissemination barrier produced the least overhead (See Table 2 for summarized results).

The next microbenchmark tests the time taken to execute a parallel directive, including the barrier which is required at its termination. As before, we show overheads for 2, 4, 16, 32, 64, 128 and 256 threads. Here the results (See Fig b) were similar to the previous barrier case where the tournament barrier produced lower overheads for test cases with 16 or more threads. The blocking algorithm continues to perform poorly (and we do not show results above 128 threads).

Table 1: Best Barrier Algorithms in the EPCC Benchmark on the SGI 3700 Altix until 256 threads and SGI 4700 Altix for 512 threads

Number of Threads	Barrier	Reduction	Parallel
2	dissemination	tree	tournament
4	dissemination	centralized	dissemination
8	dissemination	tournament	tournament
16	tournament	dissemination	tournament
32	tournament	tournament	tournament
64	tournament	tournament	tournament
128	tournament	tree	tournament
256	tournament	tournament	tournament
512	tournament	tournament	tournament

Table 2: Best Barrier Algorithms in the EPCC Benchmark on Sun Fire X4600

Number of Threads	Barrier	Reduction	Parallel
2	dissemination	tournament	tournament
4	dissemination	tournament	tournament
8	dissemination	tournament	tournament
16	dissemination	tournament	tournament

A barrier is also used as part of the OpenMP reduction implementation. The results (See Fig c) for the reduction operation are more diverse than for the other two OpenMP constructs (Barrier and Parallel). The tree implementation produced the least overheads with 2 and 128 threads, the centralized barrier performed well on 4 threads, the dissemination barrier worked well for 16 threads, and the tournament implementation worked well on 32, 64 and 256 threads. The blocking algorithm produced the worst results in all cases. On a Sun Fire X4600 and a Fujitsu-Siemens Intel Xeon running the test case with 2, 4, 8 and 16 threads, the tournament barrier produced the best results (See Table 2).

Table 1 summarizes the best algorithms for the different OpenMP constructs based on the EPCC benchmark on SGI 3700 Altix until 256 threads and SGI 4700 Altix for 512 threads. Table 2 summarizes the results on multicore systems (Sun Fire X4600 with eight dual core AMD Opteron 885 processors). It is clear that there is not a single optimal algorithm for all the different OpenMP constructs with different numbers of threads and on different platforms. The best barrier implementation depends on the environment (i.e, number of threads, system utilization) and the platform where the application is running. On a multicore system, thread binding also influenced the results. We saw an improvement in performance of these barrier implementations when the threads were bound to cores of the same processor using numactl command. These results show the need for OpenMP runtimes to be able to adaptively choose the best barrier implementation based on all these factors.

4.2 The Performance of ASPCG

We tested the barrier implementation using the Additive Schwarz Preconditioned Conjugate Gradient (ASPCG) kernel up to 128 threads on the Altix System. The ASPCG kernel solves a linear system of equations generated by a Laplace equation in Cartesian coordinates. The kernel supports multiple parallel programming models including OpenMP and MPI.

Figure 2 shows the timings of the ASPCG kernel using the different barrier implementations. Note that the blocking algorithm does not scale after 32 threads. This is because the wake-up signal to the threads becomes a contention point. All busy-wait algorithms scale, some with better performance than others. Using the dissemination implementation instead of the original barrier implementation in OpenUH, centralized blocking, represents a performance gain (total wall clock time) of 12 times for 128 threads. Table 3 shows a summary of the best barrier implementations for ASPCG on different numbers of threads.

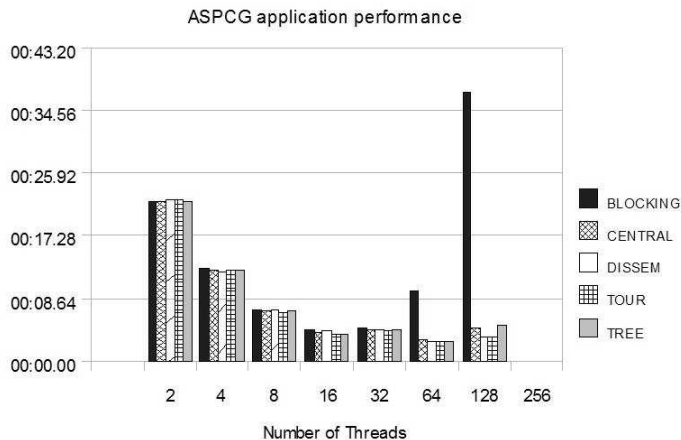


Figure 2: Timings for the ASPCG kernel with different barrier implementations

4.3 The Performance of GenIDLEST

Generalized Incompressible Direct and Large-Eddy Simulations of Turbulence (GenIDLEST) solves the incompressible Navier-Stokes and energy equations and is a comprehensive and powerful simulation code with two-phase dispersed flow modeling capability, turbulence modeling capabilities, and boundary conditions to make it applicable to a wide range of real world problems. It uses an overlapping multi-block body-fitted structured mesh topology in each block combining it with an unstructured inter-block topology. The multiblock ap-

proach provides a basic framework for parallelization, which is implemented by SPMD parallelism using MPI, OpenMP or a hybrid MPI/OpenMP. GenIDLEST uses OpenMP *for* and *reduction* constructs extensively in its code. In GenIDLEST, the centralized blocking implementations works better than the rest of the barrier implementations for 2 and 4 threads. Table 3 shows a summary of the best barrier implementations for different numbers of threads. It is clear here that the choice of a good barrier implementation can be application dependent. The use of the tournament barrier algorithm instead of a centralized blocking one represented a performance gain in total wall clock time of 35% on 32 threads.

Table 3: Best Barrier Algorithms for ASPCG and GenIDLEST

Number of Threads	ASPCG	GenIDLEST
2	tournament	blocking
4	dissemination	blocking
16	tournament/tree	dissemination
32	tournament	tournament
64	tournament	-
128	dissemination	-

5 Conclusions and Future Work

We have presented the impact of a number of different barrier implementations, including a centralized blocking algorithm and several kinds of busy wait algorithms, on the overheads of OpenMP constructs.

We implemented the barrier algorithms in OpenUH and obtained a significant reduction in overheads for barriers, and constructs that include them, using distributed busy-wait approaches. The ASPCG and GENIDLEST applications were compiled using the enhanced OpenUH system, also with noticeable performance improvements under the new busy-wait algorithms. In general, the performance of a given barrier implementation is dependent on the number of threads used, the architecture (memory layout/interconnect), application and possibly system load. An OpenMP runtime library should therefore, we believe, adapt to different barrier implementations during runtime.

Our future work includes further testing of these algorithms on larger systems (in particular, the Altix 4700 deployed at Nasa Ames with 2048 cores) and on other applications. We also plan to explore the use of similar enhancements to further improve the performance the reduction operation (i.e. updates on the reduction variable). Also as OpenMP 3.0 becomes available in OpenUH we would like to evaluate how these algorithms affect the performance of the tasking feature, especially the untied tasks and the environment variable `OMP_WAIT_POLICY`.

6 Acknowledgments

We want to thank Dr. Edgar Gabriel for providing us useful information about MPI barrier implementations in OpenMPI and Cody Addison for providing us preliminary results that motivated this work. We also want to thank the Center for Computing and Communication, at RWTH Aachen University for letting us use their cluster. This work is supported by NSF grants CCF-0833201 and CCF-0702775 (see <http://www2.cs.uh.edu/hpctools/darwin/>). The pseudocode published by the Computer Science Department of the University of Rochester was adapted for the work described in this paper.

References

- [1] C. Liao, O. Hernandez, B. Chapman, W. Chen, and W. Zheng. Openuh: An optimizing, portable openmp compiler. In *12th Workshop on Compilers for Parallel Computers*, January 2006.
- [2] Open64. <http://open64.sourceforge.net>, 2005.
- [3] John M MellorCrummey and Michael L Scotty. Algorithms for Scalable Synchronization on SharedMemory Multiprocessors. In *ACM Transactions on Computer Systems*, January 1991.
- [4] Jelena Pjeivac-Grbovic Thara Angskun George Bosilca Graham E. Fagg Edgar Gabriel Jack J. Dongarra. Performance analysis of MPI collective operations. In *Cluster Computing*, 2007.
- [5] Mihai Burcea and Michael J. Voss. A Runtime Optimization System for OpenMP. In *WOMPT*, 2003.
- [6] Jehoshua Bruck, Ching-Tien Ho, Shlomo Kipnis, Eli Upfal and Derrick Weathersby. Efficient Algorithms for All-to-All Communications in Multipoint Message-Passing Systems. In *IEEE transaction on parallel and distributed systems, vol8*, Nov 1997.
- [7] J. M.Bull and D.O Neill. A microbenchmark suite for OpenMP 2.0. In *Proceedings of the Third European Workshop on OpenMP* 2001.
- [8] D.K. Tafti. Genidlest - a scalable parallel computational tool for simulating complex turbulent flows. In *Proceedings of the ASME Fluids Engineering Division* Nov 2001.
- [9] Extreme OpenMP <http://www.cs.uh.edu/xomp/>
- [10] G. Wang, D. Tafti. Memory Systems for Solving Large Sparse Linear Systems. In *International Journal of High Performance Computing applications* 1999.