An Optimized Reduction Design to Minimize Atomic Operations in Shared Memory Multiprocessors

Abstract

Reduction operations play a key role in modern massively data parallel computation. However, current implementations in shared memory programming APIs such as OpenMP are often cause of computation bottlenecks due to the high number of atomic operations involved. We propose a reduction design that takes advantage of the coupling with a barrier synchronization to optimize the execution of the reduction. Experimental results show how the number of atomic operations involved is dramatically reduced, which can lead to significant improvement in scaling properties on large numbers of processing elements. We report a speedup of 59.64% on the *312.swim_m* SPEC OMP2001 benchmark and a speedup of 24.89% on the *streamcluster* benchmark from the PARSEC suite over the GCC *libgomp* baseline.

1. Introduction

The rise of multi-core architectures in recent years has led to the widespread need for parallel software. Given the limited improvements in clock rates, exploiting parallel execution is needed to guarantee performance improvements.

Parallelism can be exploited at several levels of granularity, from instruction level parallelism to data parallelism to task parallelism. The OpenMP [1] Application Programming Interface (API) aims at providing an easy-to-use way to program parallel applications at multiple levels of granularity, implemented on top of the C and Fortran languages. Specifically, it targets data and task parallelism by providing directives to identify parallel regions of code and parallel loop constructs.

OpenMP also offers a *reduction* clause to provide some support for recursive array computation, inspired by the *reduce* or *fold* constructs of functional languages [6].

In functional languages such as Lisp or Haskell, *reduce* is a higher-order operator that takes as input a binary function f, a list l and an initial value v, and is defined recursively as follows:

$$reduce(f, l, v) = \begin{cases} v & \text{if } |l| = 0, \\ reduce(f, l[1:], f(v, l[0])) & \text{if } |l| > 0 \end{cases}$$

If the binary function f is associative, it is possible to parallelize the reduce operation, executing it in approximately $log_2(|l|)$ steps, where each step i computes a set of intermediate results t_i by applying f to pairs of values of t_{i-1}.

OpenMP support for *reduce*-like constructs is limited to associative and commutative binary operators and, in the case of Fortran, intrinsic procedures, which are also associative and commutative functions. Arbitrary functions f are not supported.

Reduce-inspired constructs are essential for the expression of data parallelism, as they provide the means to express the extraction of synthetic results from large amounts of data. Recent works in the field of distributed computing [9] show that many data parallel computations can be easily expressed in terms of a reduce-like construct paired with a *map*-like construct. A map construct essentially allows the execution of a given n-ary function on all the n-uples obtained by taking an element from each of n sequences of equal length.

In OpenMP, the parallel loop construct provides the basic data parallelism, replacing the functional map with a somewhat more general procedural construct, but still replicating a typical map-reduce structure.

The parallel loop implements a *fork-join* model, which requires a single implicit synchronization. In the general case, a single barrier synchronization is needed for ensuring that all iterations of a parallel loop are completed at the join point before moving to other parts of the program. This implicit synchronization can be removed with a `nowait` clause, while explicit synchronizations can also be used to handle data dependencies.

On the other hand, the reduction step, which always takes place at the end of a parallel loop, requires more synchronization. This synchronization overhead leads the reduction step to cause loss of scalability, to the point where reduction overhead can become a critical issue, as shown in [11] for the *312.swim_m* SPEC OMP2001 benchmark.

The goal of this paper is to introduce an optimized barrier synchronization and reduction step, by allowing the intermediate values of the reduction to be carried along by the inter-thread communication required for the barrier synchronization.

The proposed solution is demonstrated by means of both OpenMP and pthread-based implementations. The pthread implementation is stand-alone and introduces a reduction construct.

For OpenMP, we replace *libgomp*[1] barrier synchronizations involved in a reduction with a tournament barrier [15], which is both more efficient and scalable, and mirrors the tree structure of the parallel reduction. We then use the atomically-accessible flags of the tournament barrier to store partial reduction values, thus removing the need for locks in communicating the partial values.

The rest of this paper is organized as follows. Section 2 introduces the background on barrier synchronization and reduction. Section 3 provides a detailed description of our solution, while Section 4 shows its worth through an experimental campaign on both benchmark applications and synthetic micro-benchmarks. Finally, Section 5 provides comparison with the state of the art in reduction

[1] *libgomp* is the OpenMP runtime implementation provided by the GNU GCC compiler [12].

optimization and Section 6 draws some conclusions and highlights future research directions.

2. Background

In this Section, we review the background in barrier synchronization algorithms and parallel reduction implementation, with an eye to the implementation of both features in OpenMP.

2.1 Barrier Synchronization

Barrier synchronization overheads account for a large fraction of the communication time in parallel/concurrent applications.

Barriers can be used with both message passing and shared memory programming models. In this paper, we will describe barrier algorithms in terms of the shared memory programming model, since it is the one implemented in OpenMP.

The goal of an optimized barrier algorithm is in both cases to minimize the communication involved during each barrier operation. In the case of message passing, this is represented by the packets sent, while for shared memory the communication is obtained through the execution of atomic instructions, as their execution is guaranteed to be correctly observed by threads other than the one performing them.

The minimization of barrier synchronization overheads has been addressed by a large number of studies [21] proposing new barrier algorithms. In general terms, we can identify three class of barrier algorithms: *centralized*, *dissemination* and *tree* barrier.

The centralized barrier class includes the central counter barrier [10], used in *libgomp*; the butterfly barrier [4] belongs to the dissemination class; the tournament barrier [15] is an example of a tree barrier. A full analysis of the state of the art is beyond the scope of this paper, but a good survey can be found in [21].

Distributing the barrier state among threads is a mandatory feature in the message passing programming models – it allows to distribute the communication traffic. However, it is also important in the shared memory programming model, as it allows to reduce the number of invocations of the cache coherency protocols.

2.2 Reduction Implementations

A reduction operation computes a scalar value as a combination of values in a sequence. In a OpenMP parallel region, a reduction is almost always followed by a barrier operation. This allows the reduction value to be correctly seen by all threads after leaving the barrier.

The reduction itself can be executed in several different ways. In the most trivial scheme, the reduction is computed by the master thread between two barrier operations. The reduction is computed sequentially. The first barrier ensures that the master thread sees a consistent state of the memory – all other threads must have finished the previous phase – before starting aggregating values. The second barrier blocks other threads until the reduction is completed. Such a simple scheme obviously sacrifices all opportunities for parallelization, and involves two barrier synchronizations, but the reduction itself is computed without performing any read-modify-write atomic instruction.

In general, however, the OpenMP compiler parallelizes the reduction. In this scenario, the reduction value is a variable shared among all threads. Each thread performs a partial reduction over private data, and then safely aggregate the partial reduction value to the global one. In addition to parallelization, this scheme allows the elimination of the first barrier. On the other hand, the global aggregation can be performed inside a critical section, or be executed through an atomic read-write-modify instruction – both of which are expensive.

If the hardware architecture supports fast barrier synchronization, it is also possible to perform reductions in a logarithmic number of steps, using a divide et impera approach with barriers to separate each step from the following. Since this implementation requires $log_2(n)$ barrier synchronizations, where n is the size of the sequence, it is only acceptable when there is hardware support for fast barriers.

Figure 1 reports an example for each of the three implementations, using primitives from the *libgomp* runtime [12]. The first example, Figure 1(a), shows a simple serialized implementation, while the second, Figure 1(b) reports the code generated by GCC to implement a reduction associated with a `omp for` directive. The loop boundaries `lw` and `up` are set by the OpenMP runtime and ensure that accesses to the `input` array are orthogonal between threads. Finally, Figure 1(c) performs a logarithmic reduction. Note that the code is more complex than previous examples and could be further optimized.

2.3 Atomic Operations

To allow threads to coordinate their execution, modern microprocessors support atomic memory access operations. In some cases, the atomicity is guaranteed by hardware properties for memory read and write operations. For example, on the Intel x86 P6 family processors every load and store aligned to 8/16/32/64 bits fitting into a cache line is atomic [18].

However, in most cases the atomic operations are more complex than simple reads or writes. The two most popular classes of atomic operations are the *read-modify-write* and the *compare-and-swap*.

Atomic read-modify-write instructions atomically read a value from memory, perform an arithmetic or logic operation, and write the result in the same memory address from which the operand was read. On modern microprocessors, the atomicity is implemented on top of the cache coherency mechanism [14].

Compare-and-swap instructions allow to atomically read a value from the memory, and optionally replace it with the content of an operand. Compare-and-swap operations are more powerful than any atomic read-modify-write instruction [16], but costs are comparable – in both cases, the time spent achieving atomicity is the dominant cost factor.

3. Combining Barrier and Reduction

To mitigate reductions overhead, we can combine the execution of each reduction and its associated barrier. This allows to pay synchronization cost once, while performing two operations – reduction and barrier.

To improve performance, we also aim at reducing the usage of atomic read-modify-write instructions as much as possible. Thus, we choose *tournament barrier* [15] as a starting point for our reduction design, since it achieves synchronization without performing any atomic read-modify-write instruction.

3.1 Tournament Barrier

The tournament barrier employs a binary tree data structure, where each of the threads that need to be synchronized is statically associated to an arbitrarily chosen leaf. Thus, for synchronizing n threads, the algorithm uses a tree with $2n - 1$ nodes. The algorithm operates in $log_2 n$ rounds.

The nodes of the barrier tree are partitioned into three sets: *active*, *passive* and *root* nodes. The root set contains only the tree root, while each pair of siblings nodes is composed by an active and a passive node. The active and passive nodes are arbitrarily chosen within the sibling pair.

Figure 2 reports the basic `tournament\_barrier` algorithm. The `tournament\_barrier` procedure manages the entrance of each thread into a leaf node of the barrier tree, then relies on the `walk_tree` procedure to perform the recursive traversal of the tree.

```
GOMP_barrier();
tid=omp_get_thread_num();
if(tid==0)
{
  red=0;
  for(i=0; i<SIZE; ++i)
    red+=data[i];
}
GOMP_barrier();
```
(a) Serialized

```
private_red=0;
for(i=lw; i<up; ++i)
  private_red+=data[i];
atomic_add(&red,
           private_red);
GOMP_barrier();
```
(b) Parallelized

```
GOMP_barrier();
tid=omp_get_thread_num();
sred[tid]=data[tid];
for(i=1; i<SIZE; i*=2) {
  if(tid%(2*i)==0)
    sred[tid]+=data[tid+i];
  GOMP_barrier();
}
if(tid==0)
  red=sred[0];
GOMP_barrier();
```
(c) Hand-written

Figure 1. Comparison of reduction implementations. In the serialized version the reduction is performed by the master thread, and constrained by two barrier operations. The parallelized version distributes the computation, eliminating the need of the first barrier, but paying the cost of an atomic read-modify-write atomic instruction. Finally, the last fragment performs a logarithmic reduction. This requires $\lceil log_2(n) \rceil$ barrier synchronizations, and is only worth doing when the hardware provides a fast implementation of this construct.

Procedure: $tournament\_barrier$

Require: a thread identifier tid
Require: a tournament barrier tree $tree$
Ensure: thread tid waits other before leaving the barrier
1: $leaf \leftarrow tid\_to\_leaf(tree, tid)$
2: $tsense \leftarrow load\_sense\_from\_tls()$
3: $walk\_tree(leaf, tsense)$
4: $store\_sense\_to\_tls(\overline{tsense})$

Procedure: $walk\_tree$

Require: a tournament barrier tree node $node$
Require: a sense $sense$
1: **if** $node$ is active **then**
2: $sibling \leftarrow get\_sibling(node)$
3: $parent \leftarrow get\_parent(node)$
4: $wait(sibling, sense)$
5: $walk\_tree(parent, sense)$
6: $signal(sibling, sense)$
7: **else if** $node$ is passive **then**
8: $sibling \leftarrow get\_sibling(node)$
9: $signal(sibling, sense)$
10: $wait(sibling, sense)$
11: **end if**

Figure 2. Tournament barrier algorithm. The entry point, procedure $tournament\_barrier$, wraps the $walk\_tree$ kernel, a recursive procedure that visits the tournament barrier tree.

A thread entering a passive node (`walk_tree`, line 7) signals the thread entering its active sibling that it has reached the barrier (lines 8-9), then waits for synchronization by spinning on a private flag (line 10). A thread entering an active node waits for the thread associated to its passive sibling (`walk_tree`, line 4), then it moves to the parent node, thus entering a new round of the synchronization algorithm (line 5). When it completes the synchronization algorithm, the thread in the active node notifies the synchronization to its passive sibling by setting the private flag (line 6).

By construction, the root node is reached by a single thread, and only when all threads have reached the barrier and are spinning in some passive node. When the root is reached, the exit phase is initiated, with notification of the barrier completion being propagated.

Before leaving the barrier, each thread reverses its private sense flag, to reuse the same barrier for the next synchronization.

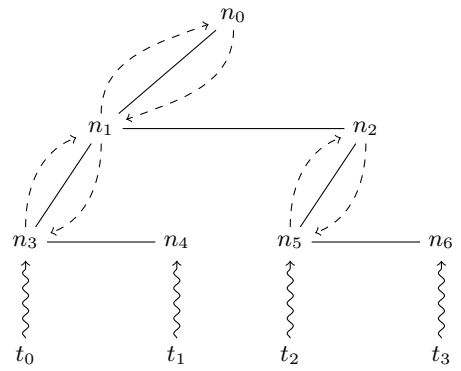

Figure 3. Execution of the tournament barrier algorithm. Each thread enters into the barrier via a statically assigned leaf. The dashed path is followed by threads entering in an active node. They climb the tree until a passive node or the root node is reached.

Example 1. Consider the four threads and the associated barrier tree shown in Figure 3. The barrier tree is a complete binary tree, with four leaves, n_3 to n_6. Odd numbered nodes are *active*, while even numbered ones are *passive*, except for the root node n_0. At the beginning each thread is assigned to a leaf node. Threads t_0 and t_2 enter into active nodes and start spinning until they are signalled by their siblings. Threads t_1 and t_3 enter passive nodes, signal their siblings, and start spinning until they are notified during the exit phase. Once t_0 and t_2 have been notified by t_1 and t_3, they move to n_1 and n_2 respectively, starting a new synchronization round. In this round t_0 is into an active node, while t_2 is in a passive node. Thus t_0 progresses to the root node n_0, while t_2 waits spinning. Once t_0 reaches the root node, it starts the barrier exit phase. First t_0 returns to n_1 and signals to t_2 to leave the barrier, then it moves to n_3, signals t_1 that synchronization has been performed and leaves the barrier; t_2, in turn, notifies t_3. Once notified, t_1 and t_3 leave the barrier.

The standard tournament barrier allows avoiding atomic read-modify-write instructions by exploiting point-to-point synchronization – each node contains a flag variable, which is written only by its sibling. Thus, each flag variable is only written by a single thread and hence no conflicts can occur.

However, such feature comes at a cost – the tournament barrier consumes more memory than other barrier algorithms. Moreover, the size and alignment of the flag must be carefully chosen to avoid

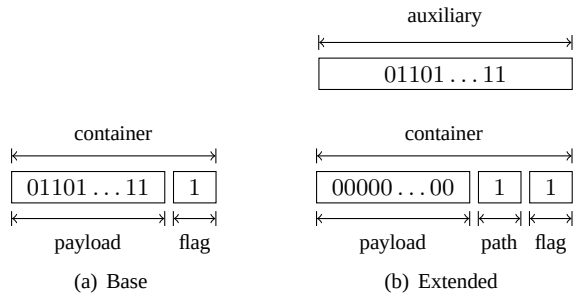

Figure 4. Layout of the container type. In the base version only one bit is needed to encode the barrier state, all other can be used to pack partial reduction values. The extended layout uses one more bit to find whether the reduction partial value is packed into the container payload or stored in the auxiliary variable.

false-sharing – indeed, if two flag variables share the same cache line, every update to one of the two triggers the execution of the cache-coherency algorithm, thus degrading performance. The flag must thus have a size equal to the cache line, even though it only carries one bit of information – all other bits are just padding. E.g., on a machine with a 64-byte (512-bits) wide cache line, each flag includes 511 bits of unused padding.

3.2 Basic Reduction Design

The key idea of our design is to exploit the free space available in the tournament barrier flag variable to propagate the partial results of the reduction operation, computing them within the nodes.

To this end, flag variables are stored into the widest type that allows atomic read/write access without locking – we will call this type the *container type* in the rest of the paper. They are also aligned to the cache line size, to avoid false-sharing.

The container type is split into two sections, shown in Figure 4(a): *flag bit* stores the state of the barrier operation (1 bit); *payload* stores the state of the reduction operation ($n-1$ bits, where n is the size of the container type).

In the case of a 64-bit machine with a 64-byte wide cache line, the container type is a 64-bit integer, aligned to 64-bytes.

As depicted in Figure 4(a), the first bit is the flag bit, while the remaining bits of the container type represent the payload.

All the bits needed to align the container to the cache line are wasted, since we cannot access them atomically without using locks and thus adding an overhead that would prevent the algorithm from achieving a speedup with respect to existing designs.

A thread entering a passive node stores into its active sibling both the flag bit and the payload containing its own partial reduction result. Then, it waits to be notified by its active sibling by spinning on its own flag variable. At each spin, the value of the flag bit is extracted from the container and checked.

A thread entering an active node first spins over its flag variable, waiting for the thread associated with the passive sibling node to reach the barrier. At each spin, the container flag variable is read and the flag bit is extracted and checked. If the flag bit is set, the payload is also extracted and aggregated with the private partial result of the thread. The thread then enters the parent node, starting a new round of the algorithm.

When a thread returns to an active node after visiting its parent, the same operations are performed as in the exit phase of the basic tournament barrier algorithm. The thread notifies its passive sibling that synchronization has been achieved by setting the flag bit into its flag variable.

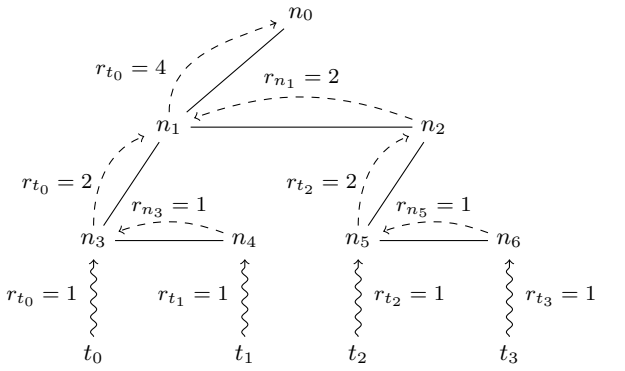

Figure 5. An example of reduction. There are four threads, each proposing 1 as the value to be aggregated. The reduction operator is sum. The r_{t_i} variable refers to the partial reduction seen by thread t_i, while r_{n_i} is the value of the partial reduction inside node n_i. The reduction is computed along the dashed path. Partial reductions are computed while moving from a node to its parent. Passive nodes sends their partial reduction values to associated active nodes.

In our design, reaching the root node has a double meaning: not only all threads have reached the barrier, but the reduction is also computed, and its value is stored in the current thread private memory space. To make this value readable to all threads, it is necessary to store it in a global-accessible variable and then force a memory fence operation. At this point the reduction is completed and the tournament barrier algorithm can proceed, notifying threads that synchronization has been achieved.

Example 2. We want to compute the sum of a sequence of unsigned integer values. Assume that the sequence to be reduced has been split into four subsequences, and partial aggregate values have been computed by each of the four threads, as shown in Figure 5. Each thread t_i ($i \in [0:3]$) enters the barrier carrying a partial aggregate value p_i. The algorithm performs the same steps as in the standard tournament barrier implementation show in Figure 3. In addition, at each step threads in passive nodes pack their partial aggregate value together with the flag value into their sibling node. Therefore in the first step t_1 and t_3 store their partial values p_1 and p_3 into nodes n_3 and n_5. Then, t_0 and t_2 before moving to the second step extract from their respective containers the payload and compute new partial values by aggregating respectively p_0+p_1 and p_2+p_3. In the second step t_2 packs its computed partial value into n_1, where it is extracted by t_0 and combined to obtain the global reduction value $p_0+p_1+p_2+p_3$. This value is published by t_0 when it reaches the root node n_0. The exit phase is unmodified with respect to Example 1.

3.3 Fast Path Optimization

The basic reduction design represents a *fast execution path*, which is only semantically correct under the condition that the reduction data-type fits the size of the container payload. To handle the remaining cases, a fall-back *slow path* will be introduced in Section .

The efficiency of the fast path strictly depends on the ability of the base tournament barrier algorithm to parallelize the reduction operation as well as to minimize the number of atomic operations. The reduction parallelism is achieved by exploiting the hierarchical structure of the barrier tree, while independence derives from limiting the entities performing the partial reductions to two, namely reader and writer. Thus an *f-way tournament barrier* [13] would not be as effective as a base algorithm for our purpose.

Procedure: $path\_management$

Require: a partial reduction value $data$
Require: a passive tournament barrier node $node$
Ensure: reduction information is communicated to the active sibling of $node$

1: $sibling \leftarrow get\_sibling(node)$
2: **if** $fits(data, payload\_size)$ **then**
3: $\quad sibling.container \leftarrow pack(data, FAST\_PATH, flag)$
4: **else**
5: $\quad siblings.auxiliary \leftarrow data$
6: $\quad mfence()$
7: $\quad sibling.container \leftarrow pack(0, SLOW\_PATH, flag)$
8: **end if**

Figure 6. Path management algorithm: when the partially reduced value fits the payload, the fast path is taken; otherwise a slow path involving a memory fence is triggered.

The fast path requires only one memory fence. The thread that reaches the root node performs this memory fence to make the final result of the reduction visible to all threads.

While the ability to take the fast path is dependent on the reduction data type, it is independent from the operator used to aggregate values. As long as partial reduction values fit into the container payload, atomic read-modify-write operations and memory fences can be avoided.

3.4 Slow Path Management

The *slow path* is designed as an extension of the basic reduction algorithm to handle the case when the reduction data-type does not fit the container payload.

To this end, the container layout has been further modified, as shown in Figure 4(b), to reserve space for a 1 bit field – the *path* field. Consequently, the payload field is shrunk by 1 bit. An auxiliary variable is added to the node state to hold the partial reduction value.

Figure 6 shows the pseudo-code of the path management algorithm. When the thread in the passive node needs to propagate the reduction value to the thread associated to the active sibling, the management algorithm is invoked. If the partial reduction does not fit into the payload (line 2), it is stored into the auxiliary variable (line 5) associated with the active sibling of the current node. Then a memory fence is issued (line 6). Finally the `flag` and `path` fields of the container are set.

Correspondingly, active nodes detect where to read the reduction partial value by reading the path bit of their container. If the path bit is set, the slow path has been taken, and the reduction partial value can be found in the auxiliary variable associated with the active node. Otherwise, the fast path has been executed – the reduction partial value is packed into the payload (line 3).

Note that the memory fence is necessary to guarantee that the partial reduction value is stored into the auxiliary variable before the flag and path bits are set, but induces an increased latency. Such fence instructions are not issued on reductions performed using the fast path, since in this case the partial reduction values and the flags are written atomically.

Since modern processors are usually 64-bit based, the payload is large enough to hold partial reduction values of most native scalar data-types. Therefore the slow path is rarely taken. In the next Section, we show how to deal with larger data types and still benefit from the fast path.

3.5 Compact Data Representation

Taking the fall-back slow path is not always necessary when the data size is too wide by just 2 bits. As an example, consider a reduction over 32-bit unsigned integers on a 32-bit machine. Since we use 1 bit to represent the flag and 1 for the path field, the payload is not wide enough to store a 32-bit unsigned integer. Thus, in the many cases where the values involved in the reduction never exceed $2^{30} - 1$, we could still use the fast path – the same might not be true in the case of signed integers, though.

The packing function used to store the partial reduction values into the payload is therefore parametrized with respect to the reduction data type and values. When working with the widest unsigned integer type that allows atomic read/write access, the packing function checks whether the value can actually fit into the payload (i.e., the two most significant bits are not set).

In these cases, the algorithm is not forced to take the slow path over all nodes – path selection strictly depends on the actual value of the reduction in each active thread. If a partial reduction value follows a slow path, this does not force a slow path for the other threads. In many cases, such as when a reduction is used to sum partial counters, it is more likely to overflow payload bounds only in the last rounds of the algorithm, which also involve only few threads, thus using a fast path in most nodes of the barrier tree.

To exploit this path optimization in the very common case where reductions are performed over word-size floating point values, we need to recover two bits from the floating point representation, without losing precision. IEEE double precision floating point numbers [17] fp are represented over 64 bits, $\langle fp_{63} \dots fp_0 \rangle$, with the following interpretation: $sign = fp_{63}$ holds the sign, $exp = \langle fp_{62} \dots fp_{52} \rangle$ represent the biased exponent, and all other bits hold the $mantissa$ (except the first digit, which is implicitly set at 1). Thus fp represent the floating point number $(-1)^{sign} \times 2^{exp-1023} \times (1.0 + mantissa)$.

To preserve precision, the algorithm cannot simply discard the least significant bits of the mantissa. We therefore operate in the same way as for the integers, assuming implicit values for two bits. These bits, and the relative assumed values, must be chosen to maximize the execution frequency of the fast path.

The distribution of mantissa bits is hard to predict, and making the sign implicit would limit the fast path to just positive or negative values. Thus, we have to choose two bits from the exponent. Since the exponent is biased, the first two bits of the exponent partition the space of floating point numbers in four equally sized subspaces. The 10_2 subspace contains exponents ranging from 1 to 512, making it a good candidate for the assumed value. The 11_2 subspace represents very large numbers (in modulo), that are expected to appear late if at all in the reduction, while 00_2 represents very small values, which would often be overshadowed by larger values early in the partial computations. Finally, the 01_2 subspace contains exponents between -511 and 0, which makes it an excellent candidate, since it represents most of the range $(2, -2)$ (excluding the values with a modulo close to zero), which is suitable for many computations.

In the end, the choice between 01_2 and 10_2 mostly depends on the application domain. For the experiments reported in this paper, we use the 01_2 setup.

3.6 Nowait Reductions

Sometimes, it is necessary to aggregate different variables at a synchronization point, and there are no data dependecies among the different reduction operations. In the case of multiple consecutive reductions, we could still use a combined reduction/barrier operation for each reduction operation. However, this scheme enforces some useless synchronizations, as once a thread has reached a passive node and has sent its reduction partial value to its active sibling,

it is no longer necessary to wait at the barrier, as synchronization is not actually needed except in the last reduction. We call this kind of reductions *nowait reductions*.

Nowait reductions are easily expressed within OpenMP programs. Work-sharing constructs, like `omp for`, can be tagged with the `nowait` clause to avoid a barrier operation before leaving the construct. If a `reduction` clause is also present, our combined barrier algorithm can be executed in nowait mode to compute the reduction value, issuing fewer atomic instructions than standard implementation.

The base algorithm has been modified to support nowait reductions. A thread t_i reaching a passive node sends the reduction partial value to its sibling t_j, and starts waiting for a synchronization achieved signal. Once notified, thread t_j performs the local aggregation pass, and then releases thread t_i *before* moving to the parent node. This allows t_i to leave the barrier earlier with respect to the base algorithm, and the exit phase is not performed at all.

This scheme keeps into the barrier only those threads that are actually working to compute partial values of the reduction, while all other can proceed to the next program statement. When the following statement is also a combined barrier/reduction operation, reductions are pipelined.

When operating in nowait mode, the thread reaching the tree root does not issue any memory fence, since synchronization is not needed, and so publishing the reduction global value is not mandatory. Consequently, if global synchronization is needed, the last barrier operation cannot be performed in nowait mode.

4. Experimental evaluation

Even though reduction is a common operation in many real world applications, there are few benchmarks designed to measure its performance. Even suites specifically designed to benchmark OpenMP implementations, such as SPEC and NAS include only a few programs that include reductions, and even fewer where reduction represents a large share of the computation time. Thus, we selected from each suite those benchmarks that actually contain enough reductions to make it worth optimizing them. To supplement these benchmarks, we also provide some micro-benchmarks to measure specific properties.

4.1 Benchmarks

We select a set of benchmarks from the most popular suites targeting shared memory parallel applications: SPEC OMP2001 [2], NAS [19], and PARSEC [3]. Fürlinger et al. [11] show the bottlenecks for the SPEC OMP2001 benchmarks. According to their analysis, *312.swim_m* and *310.wupwise_m* are the only benchmark in the suite where reductions have a significant impact, though *310.wupwise_m* uses complex data types, and thus is not optimisable in our framework. PARSEC and NAS also provide a single interesting benchmark each, *streamcluster* and *cg*.

In addition to evaluation on benchmark applications, synthetic micro-benchmarks are useful to analyze the performance properties of the proposed reduction design. The only well known micro-benchmark suite for OpenMP constructs is EPCC [5]. The EPCC *syncbench* benchmark is designed to stress reduction computations. Its kernel is a `omp parallel` region. However, the GCC OpenMP implementation introduces implicit barriers at both region start and end. Since the region body in the benchmark does not perform any relevant computation, GCC-induced synchronizations dominate the benchmark runtime, making it all but impossible to use it for its designated purpose. Moreover, *syncbench* does not help in understanding the behavior of the reduction design. Therefore, we provide in Section 4.4 four synthetic micro-benchmarks.

Table 1 shows the resulting benchmark set, characterized by the dynamic count of reduction operations, as well as by the type of reductions and the data types involved. The set covers all the interesting data types: integers and floating point numbers, in the latter case including both single and double precision.

4.2 GCC Optimization

All benchmarks except *streamcluster* are parallelized exploiting OpenMP directives. Thus, we have introduced in the GCC OpenMP compiler support for our combined barrier and reduction implementation. When a reduction clause that can be optimized is found, a GCC optimization pass identifies the barrier operations executed after the reduction, and replaces both with the invocation of our combined reduction and barrier. To this end, we have also augmented the GCC OpenMP runtime, *libgomp*, with our barrier implementation. To measure the efficiency of the combined barrier and reduction (and not the efficiency of the barrier alone), we still rely on the default barrier implementation (a central counter barrier) in all cases except those where the combined barrier and reduction is used.

4.3 Experimental Setup

The experimental campaign has been conducted on a AMD NUMA machine with four nodes, each a quad core Opteron 8378 processor. Each core has a two-level private cache hierarchy. L1 cache is composed by a 64KBytes data cache and by a 64KBytes instruction cache. L2 cache is an unified 512KBytes cache. All cores within a node share an unified 6144KBytes L3 cache. Inter-node communication is supported by a fully-connected network.

All benchmarks are compiled with the GCC 4.6 compiler in two flavors: *base* and *peak*. Base compilation is the reference execution obtained using an unmodified GCC compiler and runtime, while peak compilation applies optimization to use our combined reduction-barrier.

For each flavor, we register both the execution times, summarized in Figure 8 and the number of atomic operations performed, shown in Figure 9. All benchmarks are run with a number of threads varying from 1 to 16 – the maximum available hardware parallelism.

4.4 Micro-benchmarks

To stress our barrier implementation, we have developed four micro-benchmarks. The kernels are reported on Figure 7.

The *fast* micro-benchmark in Figure 7(a) stresses the execution of the fast path. Conversely the *slow* micro-benchmark (Figure 7(b)) always triggers the slow path. The *mixed* micro-benchmark (Figure 7(c)) evaluates the case where execution starts from the fast path and then triggers the slow path. Finally, the *multi* micro-benchmark (Figure 7(d)) targets the nowait reduction behavior in the case of multiple reductions in the same loop.

In all cases, we compare our design with the *libgomp* baseline. The results show that for the fast path and the multiple nowait reductions the number of atomic operations is very low and scales well over a larger amount of threads. However, the *multi* micro-benchmark shows that greater benefits are achieved when nowait reductions are involved since in this case our design significantly reduces the amount of synchronization, thus obtaining a major performance improvement over the *libgomp* baseline.

On the other hand, the *slow* and *mixed* micro-benchmarks show that our design does not significantly degrade performance even when the slow path is triggered.

4.5 312.swim_m

This benchmark numerically solves a shallow water modelling problem relevant to weather prediction [2]. It repeatedly executes a computationally intensive loop body containing a parallel OpenMP

Benchmark	Suite	Field	Language	Reductions	Operator	Data Type (bits)	Data Size
312.swim_m	SPEC OMP2001	Weather prediction	Fortran	2400	+	floating point	64
cg (class C)	NPB	Fluid dynamics	Fortran	3900	+	floating point	64
streamcluster	PARSEC	Online clustering	C	4235	+	unsigned integer	32
fast	Micro-benchmark		C	50000	&	unsigned integer	64
slow	Micro-benchmark		C	50000	&	unsigned integer	64
mixed	Micro-benchmark		C	50000	+	unsigned integer	64
multi	Micro-benchmark		C	50000	&	unsigned integer	64

Table 1. Benchmark characterization: dynamic count of reductions is reported, together with the operators and data types involved in the reductions.

```
acc=ULONG_MAX;
#pragma omp parallel
for(unsigned i=0; i<INPUT_SIZE; ++i)
  #pragma omp for reduction(&:acc)
  for(unsigned j=1; j<OMP_LOOPS; ++j){
    delay();
    acc&=input[i]*j&ULONG_MAX;
  }
```

(a) fast

```
acc=ULLONG_MAX;
#pragma omp parallel
for(unsigned i=0; i<INPUT_SIZE; ++i)
  #pragma omp for reduction(&:acc)
  for(unsigned j=1; j<OMP_LOOPS; ++j){
    delay();
    acc&=input[i]*j;
    acc|=OVERFLOW_MAGIC;
  }
```

(b) slow

```
acc=0;
#pragma omp parallel
for(unsigned i=0; i<INPUT_SIZE; ++i)
  #pragma omp for reduction(+:acc)
  for(unsigned j=1; j<OMP_LOOPS; ++j){
    delay();
    acc+=input[i]*input[i]*j*j;
  }
```

(c) mixed

```
acc=add=aee=ULONG_MAX;
#pragma omp parallel
for(unsigned i=0; i<INPUT_SIZE; ++i)
  #pragma omp for reduction(&:acc,add,aee)
  for(unsigned j=1; j<OMP_LOOPS; ++j) {
    delay();
    acc&=input[i]*j;
    add&=(i%2)*input[i]*j;
    aee&=((i-1)%2)*input[i]*j;
  }
```

(d) multi

Figure 7. Micro-benchmark kernel codes. The four micro-benchmarks test respectively the fast path, slow path, transition between fast and slow path and multiple nowait reductions. The `OVERFLOW_MAGIC` is a mask employed to set the two most significant bit to 1.

region. At the end of the parallel region three reductions are computed.

Our algorithm generates two nowait reductions followed by a combined reduction-barrier.

As shown in Figure 8(e) and 9(e) the number of atomic operations performed by the peak version is always lower than the baseline, while run-times are lower or equal to the baseline when working with more than 4 threads.

When working with only few threads, atomic operations are often uncontested. Thus, the base implementation can outperform the peak implementation in these cases. On the other hand, when the number of threads grows the performance of the peak optimization stabilizes and are always better than the baseline.

4.6 cg

The *cg* benchmark computes the eigenvalues of a sparse matrix using the conjugate gradient method [19], relevant to the field of computational fluid dynamics. The structure of the code includes a top-level loop that contains an OpenMP parallel region. The parallel region computes aggregate value used in subsequent loop iterations by means of a reduction at the end of the region. It is important to note that a `omp master` construct is used to compute an intermediate aggregate value, and an explicit barrier is used to block all other threads.

We have executed the benchmark using the C data set.

Even if our reduction implementation is effective in reducing the number of issued atomic operations, as shown in Figure 9(f), the run-time is dominated by the `omp master` sections, thus run-times do not scale well.

4.7 streamcluster

The *streamcluster* benchmark solves the online clustering problem [3] relevant to the field of data mining. It periodically consumes a set of data items that are processed in parallel. An aggregate value is computed to find potential clusters. Sets of data items are processed within a parallel region implemented by means of a set of threads. The computation is organized in phases delimited by barriers. Several reduction operations are used to compute the aggregate values.

As shown in Figure 8(g) and 9(g) the number of atomics operations performed by the peak implementations is always lower than the baseline, and the same holds for run-times.

Since the benchmark employs a monitor structure, it has inherent limits to the available parallelism. Thus performance does not scale over 6 threads.

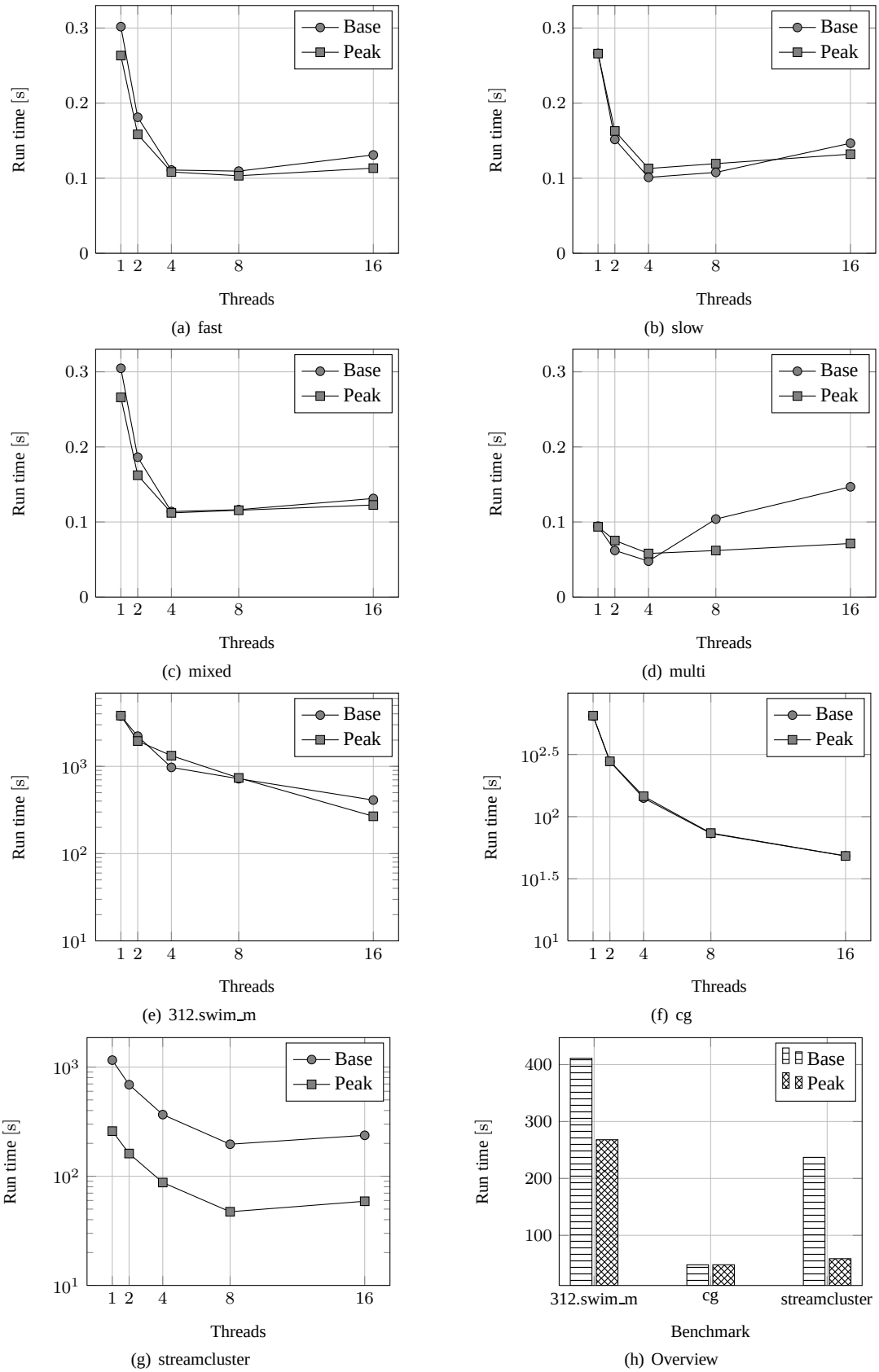

Figure 8. Plot of benchmark run-times. Each graph compares base and peak runs of a single benchmark, varying the number of worker threads. The last graphs reports run-times of the most representative benchmarks ran using 16 threads.

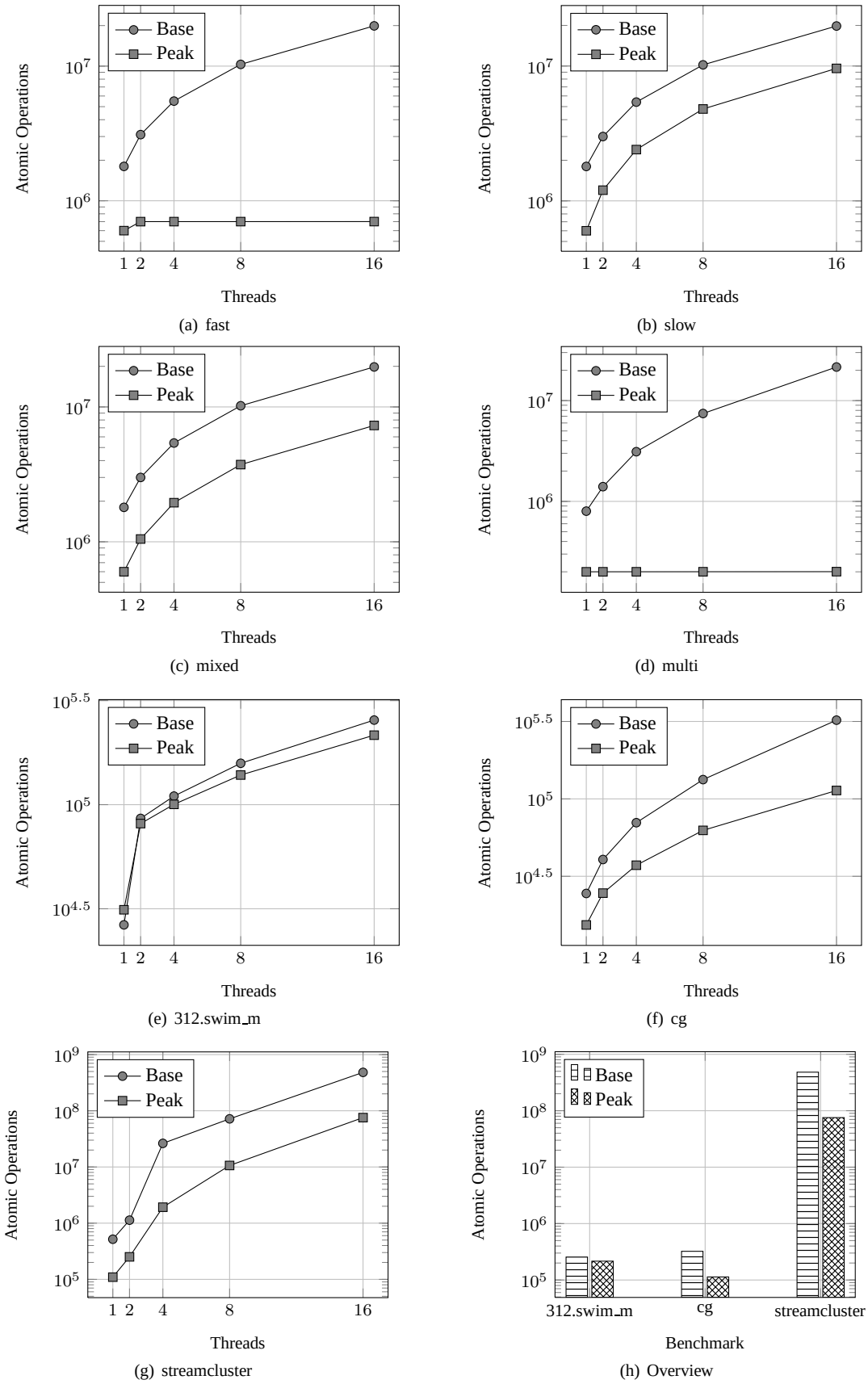

Figure 9. Number of issued atomic operations per benchmark. Each graph compares base and peak runs of a particular benchmark. The last graph summarizes the number of atomic operations issued by the most representative benchmarks ran using 16 threads.

5. Related work

Many barrier synchronization algorithms have been proposed. Nanjegowda et al. [21] provide a survey of barrier algorithms in the context of OpenMP, reporting good scalability for the tournament barrier.

In the context of distributed computing, where communication is more expensive than in shared memory architectures, the MPI standard [20] includes the notion of *collective operations* to perform multiple operations in one step, and reduce the number of exchanged messages.

Shirako et al. [24] introduces the concept of *phaser accumulator* to combine reduction and barrier operations in presence of dynamic parallelism. They rely on atomic read-modify-write instructions to safely send reduction partial values to the phaser. A tree-like structure for phasers is proposed in [23], with the goal of improving phaser scalability. With respect to our work [23] focuses on introducing a reduction capability in the X10 [7] phaser construct. The authors compare their work with the OpenMP runtime using the EPCC Syncbench [5]. This makes comparison with our work difficult both because of the characteristics of the EPCC Syncbench described in Section 4.1 and the different goals, as our work is geared towards a language agnostic reduction design.

Chun and Xuejun [8] address the optimization of barriers and reductions by a different approach – rather than handling the fast paths at runtime, they rely on new primitives for expressing constrained forms of barrier and reduction constructs.

The exploitation of time spent by threads waiting at barriers is also addressed in OpenMP [1], by means of the `omp task` directive, which allows the definition of tasks that are executed by threads while waiting for barrier synchronization.

Packing data into a native word to exploit atomic read-modify-write instructions is a well known technique in the field of locking algorithms. E.g., Russell and Detlefs [22] organize a native word to contain pointer to thread and a three shape bits. Compare-and-swap operations are then used to atomically bias a lock to a thread.

6. Conclusions

In this paper, we have proposed a reduction design to take advantage of the coupling with a barrier synchronization. Our design exploits the unused space in the flag variables of a tournament barrier to carry a partial reduction value, thus reducing the amount of atomic operations.

Our experimental campaign shows a significant reduction in the number of atomic operations employed to perform the reductions, as well as a speedup of 59.64% on the *312.swim_m* and 24.89% on the *streamcluster* benchmark.

Future directions for this research line include affinity-guided association of threads to barrier tree leaves, as well as an adaptive data-compaction method to further improve of the frequency of the fast path.

References

[1] *OpenMP Application Program Interface, version 3.0*. ARB, 2008.

[2] V. Aslot, M. J. Domeika, R. Eigenmann, G. Gaertner, W. B. Jones, and B. Parady. SPEComp: A new benchmark suite for measuring parallel computer performance. In R. Eigenmann and M. Voss, editors, *WOMPAT*, volume 2104 of *Lecture Notes in Computer Science*, pages 1–10. Springer, 2001. ISBN 3-540-42346-X.

[3] C. Bienia, S. Kumar, J. P. Singh, and K. Li. The PARSEC benchmark suite: characterization and architectural implications. In A. Moshovos, D. Tarditi, and K. Olukotun, editors, *PACT*, pages 72–81. ACM, 2008. ISBN 978-1-60558-282-5.

[4] E. D. Brooks. The butterfly barrier. *Int. J. Parallel Program.*, 15(4): 295–307, 1986.

[5] J. M. Bull. Measuring synchronisation and scheduling overheads in OpenMP. In *In Proceedings of First European Workshop on OpenMP*, 1999.

[6] M. M. T. Chakravarty, R. Leshchinskiy, S. P. Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: http://doi.acm.org/10.1145/1248648.1248652.

[7] P. Charles, C. Grothoff, V. A. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In R. E. Johnson and R. P. Gabriel, editors, *OOPSLA*, pages 519–538. ACM, 2005. ISBN 1-59593-031-0.

[8] H. Chun and Y. Xuejun. Improve OpenMP performance by extending BARRIER and REDUCTION constructs. In A. V. Veidenbaum, K. Joe, H. Amano, and H. Aiso, editors, *ISHPC*, volume 2858 of *Lecture Notes in Computer Science*, pages 529–539. Springer, 2003. ISBN 3-540-20359-1.

[9] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pages 137–150, 2004.

[10] E. Freudenthal and A. Gottlieb. Process coordination with fetch-and-increment. In *ASPLOS*, pages 260–268, 1991.

[11] K. Fürlinger, M. Gerndt, and J. Dongarra. Scalability analysis of the SPEC OpenMP benchmarks on large-scale shared memory multiprocessors. In Y. Shi, G. D. van Albada, J. Dongarra, and P. M. A. Sloot, editors, *International Conference on Computational Science (2)*, volume 4488 of *Lecture Notes in Computer Science*, pages 815–822. Springer, 2007. ISBN 978-3-540-72585-5.

[12] GNU. GNU libgomp. `http://gcc.gnu.org/onlinedocs/libgomp/`.

[13] D. Grunwald and S. Vajracharya. Efficient barriers for distributed shared memory computers. In H. J. Siegel, editor, *IPPS*, pages 604–608. IEEE Computer Society, 1994. ISBN 0-8186-5602-6.

[14] J. L. Hennessy and D. A. Patterson. *Computer Architecture - A Quantitative Approach*. Morgan Kaufmann, fourth edition, 2007.

[15] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1980.

[16] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, 1991.

[17] *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, 2008.

[18] *Intel 64 and IA-32 Architectures Software Developer's Manual*. Intel, 2009.

[19] H. Jin and M. Frumkin. The OpenMP implementation of NAS parallel benchmarks and its performance. Technical report, NASA, 1999.

[20] *MPI: A Message-Passing Interface Standard, Version 2.2*. Message Passing Interface Forum, 2009.

[21] R. Nanjegowda, O. Hernandez, B. M. Chapman, and H. Jin. Scalability evaluation of barrier algorithms for OpenMP. In M. S. Müller, B. R. de Supinski, and B. M. Chapman, editors, *IWOMP*, volume 5568 of *Lecture Notes in Computer Science*, pages 42–52. Springer, 2009. ISBN 978-3-642-02284-5.

[22] K. Russell and D. Detlefs. Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In P. L. Tarr and W. R. Cook, editors, *OOPSLA*, pages 263–272. ACM, 2006. ISBN 1-59593-348-4.

[23] J. Shirako and V. Sarkar. Hierarchical phaser for scalable synchronization and reductions in dynamic parallelism. In *IPDPS*. IEEE, 2010.

[24] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer. Phaser accumulators: A new reduction construct for dynamic parallelism. In *IPDPS*, pages 1–12. IEEE, 2009.