Highly Scalable Barriers for Future High-Performance Computing Clusters

Holger Fröning, Alexander Giese
University of Heidelberg, Germany
{holger.froening, alexander.giese}@ziti.uni-heidelberg.de

Héctor Montaner, Federico Silla, José Duato
Universitat Politècnica de València, Spain
hmontaner@gap.upv.es, {fsilla, jduato}@disca.upv.es

Abstract— **Although large scale high performance computing today typically relies on message passing, shared memory can offer significant advantages, as the overhead associated with MPI is completely avoided. In this way, we have developed an FPGA-based Shared Memory Engine that allows to forward memory transactions, like loads and stores, to remote memory locations in large clusters, thus providing a single memory address space. As coherency protocols do not scale with system size we completely avoid a global coherency across the cluster. However, we maintain local coherency domains, thus keeping the cores within one node coherent. In this paper, we show the suitability of our approach by analyzing the performance of barriers, a very common synchronization primitive in parallel programs. Experiments in a real cluster prototype show that our approach allows synchronization among 1024 cores spread over 64 nodes in less than 15us, several times faster than other highly optimized barriers. We show the feasibility of this approach by executing a shared-memory implementation of FFT. Finally, note that this barrier can also be leveraged by MPI applications running on our shared memory architecture for clusters. This ensures the usefulness of this work for applications already written.**

Keywords- computer communications, computer synchronization, high performance networking, distributed shared memory

I. INTRODUCTION

Due to cost reasons, large high performance computing deployments are usually based on leveraging many off-the-shelf nodes arranged in a cluster. However, as these nodes are independent systems, each of them has its own memory address space. In this way, parallel applications running in a cluster must be designed according to the message passing paradigm (MPI typically), which unfortunately presents higher communication overheads than the shared-memory one [1]. Additionally, message passing is less intuitive to most programmers than shared memory.

Because of the benefits of a single global memory address space, companies like SGI [2] or ScaleMP [3] have devised mechanisms that aggregate resources in a cluster into a single address space. These proposals start from commodity components in order to keep cost as low as possible. Additionally, all of them provide a coherent memory address space across the cluster by extending the original coherency protocol inside the nodes with a higher-level protocol that keeps coherency for the entire system. Unfortunately, this second protocol reduces performance and, additionally, its scalability is clearly limited, thus bounding in practice the size of these particular coherent distributed shared-memory clusters to about 128 nodes.

We recently proposed in [4] a new hardware approach for gluing resources in clusters. The main difference with respect to previous proposals is that the new architecture does not keep coherency among nodes, thus increasing performance and presenting the same scalability capabilities than large message passing clusters. Note that although the new architecture provides a single address space that is not kept coherent between nodes, the original coherency protocol among processors inside a node is not overridden. This provides a system partitioning similar to the Partitioned Global Address Space (PGAS). However, note that [4] presents a new memory cluster architecture rather than a programming model.

The new architecture is based on the observation that current processors provide coherency features that exceed the requirements of regular shared-memory applications. We improve scalability by avoiding this unnecessary effort towards sequential consistency. Instead, we rely on relaxed consistency [25], where the synchronization primitives, already present in parallel code, constitute a *safety net*, thus becoming the synchronization points where coherency is required for the correct execution of the application. Therefore, in order to leverage this distributed non-coherent shared-memory architecture for parallel applications, new synchronization primitives like barriers and locks must be provided.

In this paper, we take such a challenge and present how barriers, as an important synchronization primitive for applications spanning several nodes, in this new cluster architecture can be optimized. We show that technology improvements will not increase performance significantly. Instead, architectural optimizations yield much higher performance increase. Our barrier implementation is evaluated using micro benchmarks and by the execution of real applications. In this way, we present a shared-memory execution of a FFT on our distributed shared-memory cluster prototype, demonstrating the feasibility and scalability of our approach. This execution is relying both on our shared memory concept and on the barrier presented here. In addition, we provide an extrapolation of barrier latency for future Exascale systems. Finally, note that message-passing clusters can also benefit from our approach if their communication engines are extended to provide support for non-coherent remote loads and stores.

The remainder of the paper is structured as follows: next, a discussion about previous work on barriers is presented. Section 3 introduces our architecture, while in Section 4 the barrier implementation is described. Section 5 focuses on optimizing the barrier. Section 6 provides an extrapolation about how the proposed barrier implementation would behave in Exascale deployments. Section 7 presents a real example of the barrier used in a shared-memory based FFT execution. Finally, the last section concludes the paper.

II. Related Work

Algorithms for barriers and other global reduction techniques in message passing systems have been extensively studied and optimized [16] [17] [18] [19], as this environment has always been the traditional ecosystem for large parallel applications. Barriers are implemented either by point-to-point messages or by using more complex patterns like the global reduction. Further optimizations lead to tree-based or butterfly structures.

Some highly optimized message passing systems, like the BlueGene, offer dedicated hardware for collectives that can noticeably improve the performance of barriers [10]. However, the cost for specialized hardware modules, or even complete networks, is extremely high. Specialized message-passing silicon is also described in [15], reporting barrier latency reductions of about 10% due to the offloading capabilities of the specialized hardware. The work in [15] also addresses the current multi-core trend. In this regard, message passing is appropriate for inter-node communication, but certainly not for intra-node use. The availability of shared memory resources should be leveraged in this case. Thus, the barrier presented in [15] is highly optimized for modern multi-core systems, consisting of three phases: the first and last stages leverage shared-memory intra-node synchronizations, while the second one is used to synchronize one process per node for all participating nodes. Similarly [14] presents a shared memory bypass in the message passing stack that reduces overhead and optimizes performance for multi-core systems by 50%-60%, compared to Gigabit Ethernet back in 2007.

As can be seen, message-passing systems can support the multi-core trend by making use of shared-memory intra-node communication. The traditional approach to implement a shared-memory barrier is based on a locked counter, where mutual exclusion of multiple writers is ensured by atomic operations. Optimizations to this approach lead to the butterfly barrier [13], where the drawbacks of the locked counter, like contention and critical regions, are avoided by superimposing a tree structure. A different approach is the dissemination barrier [20], which disseminates the barrier information to groups of processes. Thus, congestion due to spinning on one single shared variable is avoided, greatly improving performance.

Even if message-passing systems leverage local shared-memory intra-node communication, the overall overhead associated with message passing is still high. To the best of our knowledge, no previous publication deals with synchronization on mixtures of coherent and non-coherent shared memory architectures, as the one we leverage in this paper. Nevertheless, we will employ techniques similar to the ones mentioned above, but with different motivations and/or outcomes.

III. System Architecture

The main goal of this work is to support scalable barrier synchronization over globally non-coherent, but locally coherent distributed shared memory clusters. The distributed shared memory pool is set up by aggregating memory from the nodes in the cluster, thus creating a global address space that spans the entire cluster. Each node within the cluster can access any arbitrary location in this global address space by using a special hardware unit, referred to as Shared Memory Engine. In our current prototype, it is located on an add-in card, which also integrates support for a custom high performance interconnection network. Off-the-shelf computers are used, as one of the goals of this new architecture is providing shared memory at low cost. In this section, we briefly present this new memory aggregation model. An extended description of the new architecture can be found in [4].

A. Address Space Layout

In the new architecture, the RAM memory installed at each of the nodes in the cluster is split into a private partition and a shared partition. The actual sizes of the private and shared partitions are irrelevant from the architecture's point of view. The aggregation of shared partitions forms the globally addressable memory (left part in Figure 1). Moreover, in order to be able to address any shared-memory location in the cluster, the shared memory pool is mapped into the virtual memory of each node. In the example shown in Figure 1, all addresses above 2^{40} refer to the global memory pool.

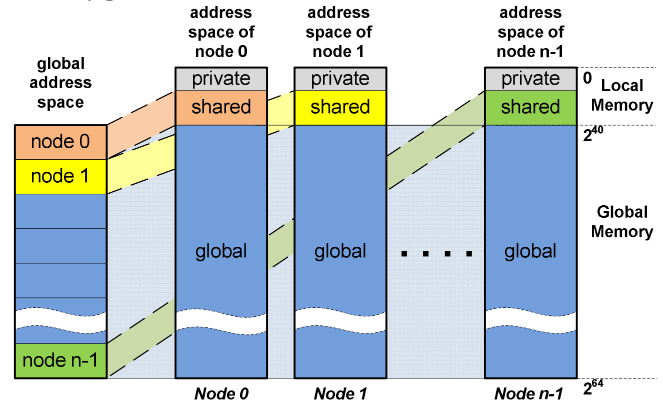

Figure 1. Address Space Layout

Processes in a given node can access remote memory locations belonging to shared partitions of other nodes with global addresses mapped to the Shared Memory Engine (obviously, leveraging both local and remote engines). Moreover, processes in one node can access the shared partition in that node either directly (just like regular RAM) or over the Shared Memory Engine as a loopback access.

In order to minimize the communication overhead, we avoid logical to physical address translations by employing

physically contiguous and pinned memory regions for the shared local partitions on all the nodes in the cluster. Using such memory regions, the hardware unit can directly access remote memory locations with physical addresses.

During system initialization, processors are configured so that memory accesses targeted to the global address space are forwarded to the Shared Memory Engine. Later, to access remote memory, CPUs execute a regular load or store instruction that results in a PIO read or write operation on a global address, mapped to the Shared Memory Engine. Global addresses are set in such a way that a part of the global address determines the target node identifier, where the network packet should be routed. The load/store operation is then encapsulated in a network packet and transferred over the network to the target node. The other part of the global address points to the location within the shared local partition of that node. In the case of a store, the operation is completed by writing the payload to this location. In the case of a load, the corresponding memory location is read and an appropriate response is sent back to the source node. This response is then forwarded to the CPU that is awaiting the response.

B. *The Shared Memory Engine (SME)*

The major task of the shared memory engine is to forward memory accesses, like reads and writes, as described above. Reads are carried out as split phase transactions, where the read request is answered by an independent packet-based response. Writes are carried out by simply writing data to the required address. Several sub-tasks can be identified based on this, like target node determination, source tag management, etc.

Latency is of paramount importance when accessing memory, in particular for remote memory locations. Due to this, we deploy a custom high performance interconnection network among nodes optimized for low latencies, in order to minimize the overall latency [6]. In addition, we employ HyperTransport (HT) to connect the SME directly to one of the host CPUs.

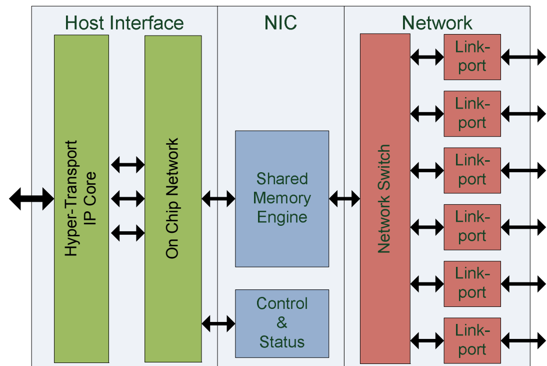

Figure 2. Topology and top-level block diagram

Figure 2 shows the block diagram of the logic implemented in the add-in card. On the left side (in green) the HT core [5] connects to the host system and to the On Chip Network [7]. The on chip network connects to the Shared Memory Engine and auxiliary units for control and status purposes (shown in blue). On the right side (in red) is the custom network switch [6] with the six optical fiber connections to the network. We use these links to set up a direct network topology, for instance 2D or 3D tori and meshes. As the switching functionality is already integrated in the add-in card, no external switches are required. For the experiments here, a 2D mesh is set up. A more detailed description of the SME can be found in [8].

C. *Prototyping*

In order to allow fast prototyping of the complete system, an existing HT infrastructure based on *Field Programmable Gate Arrays (FPGAs)* is used [9]. As reconfigurable logic in form of FPGAs is employed, the performance in terms of bandwidth is limited (624MB/s per link and 1.6GB/s for the host interface).

The SME architecture itself is very lean and efficient due to its pipelined implementation. However, several problems exist which limit the overall performance of the system. They do not origin from the SME itself, but from the behavior of the CPUs in our prototype (AMD Opteron K10 [1]). These problems come from the fact that the communication engine is only accessible over *Memory Mapped IO (MMIO)* space; hence, the host CPUs treat it like a peripheral device. Modern CPUs restrict PIO accesses to MMIO in at least two aspects: the number of outstanding load transactions is typically limited to one and the payload of such a transaction is at most 64 bits. However, these limitations do not apply for store transactions.

These problems may be overcome by moving the communication engine from MMIO space to DRAM space. Then it would be treated by the host CPU as a regular memory controller and the CPU would not restrict the accesses. However, moving it into DRAM space is a very complex step, as it requires the Shared Memory Engine to participate in the processor coherency protocol.

Another solution is used here, which tries to avoid remote reads whenever possible. Instead of such a pull-model, a push-model is followed, where the remote side "pushes" the data into the local node's main memory. This push model will be further addressed in Section 5, devoted to optimize the barrier implementation previously proposed in Section 4.

IV. BARRIER IMPLEMENTATION AND ANALYSIS

In parallel computing, a barrier is a primitive that constitutes a synchronization point in the concurrent execution of several flows where every thread/process of the parallel application must wait for the rest of threads/processes to arrive before proceeding. Depending on the memory model, barriers are usually based on the exchange of messages [10] or on atomic operations [11]. Atomic operations provide a way to serialize accesses to shared data (the barrier in our case). Thus, a barrier in shared memory can be safely implemented by a locked counter that tracks the threads that reached the barrier.

[1] Similar restrictions do apply for comparable x86-64 architectures from other vendors.

In our prototype cluster, atomic operations cannot be used, as current AMD processors do not support atomic HT transactions like Compare-And-Swap or Fetch-And-Add (although defined in the HT specification). Instead, AMD processors rely on locks to implement atomic operations [12]. However, these locks are only supported in coherency domains, thus not being suitable for our SME because it is located in MMIO space, outside of the coherency system. Therefore, a different approach is required.

A. *Implementation*

In the same way barriers in regular multi-core single-node parallel computers are carried out by software, a software solution without need for atomic operations is required to implement a barrier in our distributed non-coherent shared-memory architecture. Such solution would be typically based on avoiding multiple write accesses to a single memory location, rendering the need for atomic operations unnecessary. In other words, if the barrier data structure spreads over several memory locations, each of them being updated by only one thread, then mutual exclusions and atomic operations can be avoided. The drawback is that now several locations have to be checked, but this overhead is negligible. However, the advantage is that this distributed approach does not lead to heavy congestion while accessing the single memory location holding the synchronization data (the counter) as in the implementation above based on atomic operations. Such an approach is used here.

```
void barrier_wait ( barrier_type* barrier, int id )
{
int my_phase, i;

my_phase = *(barrier->phase);
barrier->flags[id] = !my_phase; //flip my flag

if ( !id ) { // only master
    // 1. wait for all others to reach barrier
    // (i.e., their flags are flipped)
    for ( i = 1; i < barrier->count; i++ )
        // no need to read it's own flag
        while ( barrier->flags[i] == my_phase );

    // 2. flip phase to signal 'barrier reached'
    *(barrier->phase) = !my_phase;

} else // remaining threads wait for master
    while ( *(barrier->phase) == my_phase );
}
```

The *barrier_type* struct in the code above consists of a phase variable and one flag for every thread. One of the threads taking part in the barrier is the master thread, while the other ones are referred to as slave threads. Each slave thread is only updating its *flag* variable to signal that it has entered the barrier. It does so by toggling its flag to match the next phase. Afterward, it waits for the barrier complete signaling, which is done by actively polling on the *phase* variable to toggle. The master thread observes the *flags* of all the other threads, and as soon as all the flags are changed, the barrier is reached. The master then updates the *phase* variable by toggling it (again an update by one single writer) in order to signal the slaves about barrier completion. At this point all threads (master and slaves) can proceed. As can be seen, every memory location is only updated by one single process.

The algorithm is based on two alternating phases. When all flags match the current phase, the barrier is reached. Afterward, the phase is just flipped; avoiding the need for re-initialization. On the other hand, as the global view of the shared memory is not coherent, local caching of remote variables like the flag or the phase is not possible: in case remote memory locations are cached, threads would not see any updates.

B. *Analysis*

The proposed barrier implementation is suitable for any kind of shared memory environment, independently if it is based on local or remote memory. In this way, the same implementation can be used either for regular small-scale multi-core systems or for larger deployments consisting of a large number of computing nodes, thus allowing a smooth transition.

Completely independent of this is the performance of our barrier implementation proposal. Figure 3 shows the basic performance of this software barrier, also compared to the performance of a PThread barrier. All experiments are run on quad-socket machines with 16 cores per machine at 2.2GHz. The memory interface is DDR2-667 and recent Linux is used as operating system. In each experiment, 10.000 barriers are executed. Latency numbers reported in the figure are the average time per barrier.

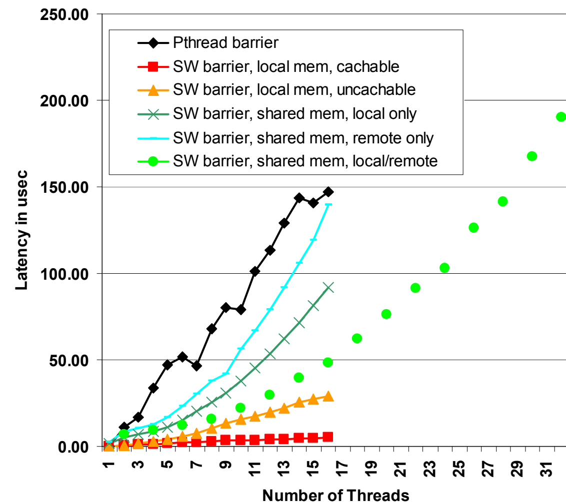

Figure 3. Unoptimized barrier performance

The PThread barrier (in blue) is the reference, as it is the current “de facto” shared-memory implementation. However, it is only applicable to local systems as it assumes support from an underlying coherency protocol that keeps all the system coherent. Interestingly, the performance of our software barrier in regular local memory (in red) is even better. For example, for 16 threads, our proposal achieves a latency of only 5.0us compared to the latency of a PThread barrier with 147.3us.

However, these numbers leverage the local coherence scheme, which is not available in our distributed shared-memory system. If the software barrier is accessed over the SME, the performance significantly drops due to the missing coherency and caching. In green, the performance is shown when all the threads and the barrier data are located in the same machine and in cyan when threads are located on a remote node (note that in the local/remote scenario, threads are distributed over two nodes). As can be seen, performance noticeably decreases with respect to the case of using local memory. This performance drop is much larger than the memory access latency increment would justify (approx. 100ns for local memory vs. 1.89us for remote memory). Thus, in addition to the memory access latency increment, the reasons for this performance difference are the following (ordered by importance):

1. Outstanding transactions: only one read operation can be outstanding per socket. This can be clearly seen in Figure 4, as performance significantly drops for more than four threads (linear increase of latency).
2. Uncacheable memory: every thread has to access the memory location of the barrier for every load or store as the use of caches should be avoided due to the lack of a coherency protocol, as mentioned previously.

The most efficient solution would be to remove the restriction of outstanding transactions, but this is a limitation introduced by the CPU manufacturer and hence not possible.

C. *ASIC performance estimation*

Although the SME is highly efficient, it adds overhead to every memory access. The latency of a local access over the SME is about 1.21us, and the latency of a remote access is about 1.89us [8]. These are outstanding numbers compared to (even ASIC based) message passing systems, but rather slow compared to main memory accesses.

If this shared-memory cluster prototype used ASIC technology instead of FPGAs, it would deliver much higher performance, as remote memory access latency would be dramatically reduced. Here we estimate how an ASIC implementation would improve performance. To do so, we base the ASIC predictions on two measurements with HT200 and HT400, and an FPGA core frequency of 156MHz. Remote memory latency can be split into several components: the time used by the source CPU, the time spent in the SME logic, and the time used by the remote memory controller. The fraction of time spent in the SME core logic is determined in simulations. Counters in the FPGA design allow measuring the fraction of time spent in the target side waiting for the memory controller (MC) access. The remaining fraction is the time spent within the source CPU. For simplicity and because of its negligible fraction, we assume that link bandwidth and delay are kept constant. On the other hand, it is expected that both the MC and the CPU fraction consist of a variable and a constant part: the variable part will scale with HT frequency, but the fixed part not. On the contrary, the complete SME core logic scales linearly with frequency. With the results from the two experiments, the constant and variable parts of both CPU and MC time can be derived by modeling the variable part in clock cycles, and the fixed part in absolute time. A solution is found if the following equation is valid for both experiments:

$$time = \frac{cycles_{fixed}}{f_{HT}} + time_{absolute} \qquad \text{(Eq. 1)}$$

Using this, predictions for higher HT frequencies and core frequencies are possible, like it will be the case for ASIC implementations. Such an implementation is expected to reach a core frequency of at least 800MHz and to implement at least an HT2000 interface. As seen in Figure 4, the full round trip latency of a load transaction decreases down to 534ns.

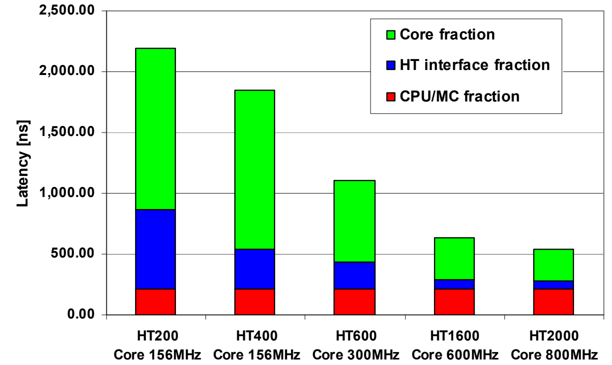

Figure 4. Prediction of remote load latency for an ASIC technology

In order to estimate the performance of the barrier for an ASIC technology, the latency prediction is applied to the number of load and store operations per barrier. By analyzing the barrier algorithm, one can find that the number of store operations is only dependent on the number of threads taking part in the barrier. Unfortunately, the number of load operations cannot be derived like this, as it is highly dependent on the relative situation in time of all threads, i.e. even a small jitter may introduce extra waiting times and thus increase the number of read operations. Thus, we rely on performance counters within the SME to determine the number of remote loads.

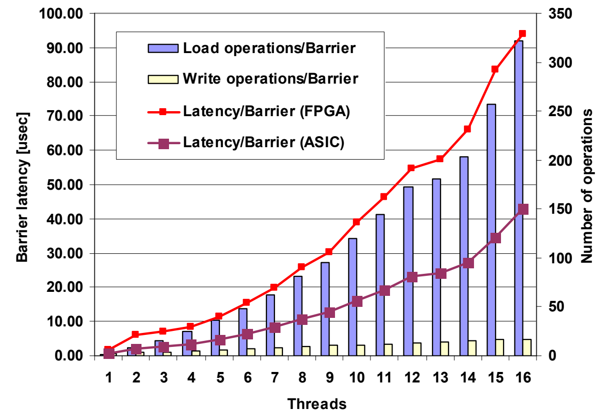

Figure 5. Number of per barrier operations with associated barrier lateny

Figure 5 reports the number of read and write operations on the right y-axis, and the corresponding barrier latency on the left y-axis. By applying the ASIC latency numbers from above one can see that the latency of a barrier for 16 threads is reduced from 94us down to 43us. This is a dramatic reduction, however the difference to the reference implementation in regular local memory is still significant (remote memory latency is now only 5 times slower than local memory latency, but the barrier performance is still reduced by a factor of 16).

V. Barrier Optimization

As the inferior performance of an FPGA compared to an ASIC does not seem to be the major reason for the performance difference, the other reasons (non-cacheable accesses and limited amount of outstanding transactions) must be addressed.

A. Use of remote store programming

It can be seen that the barrier is heavily relying on remote load operations for spinning purposes. The following considerations help to visualize the impact of this:

1. The amount of outstanding remote loads per socket is limited due to CPU restrictions. Thus, one thread may have to wait for other threads in the same socket to finish their remote loads.
2. The latency of a remote load is high compared to a local load operation. As can be seen in the previous section, this is not the main reason, but has to be taken into account.
3. Last, although caching of remote memory locations is possible while spinning, it is not suitable for synchronization purposes, as coherency is not kept. However, spinning on local memory locations does not suffer from these constraints.

The conclusion is that remote loads should be avoided where possible, in a very similar manner as in [27] or in the dissemination barrier. Therefore, instead of spinning on remote locations, we should push the new data to a location closer to the executing CPUs. This way, we cannot only avoid CPU restrictions, but also leverage the local memory hierarchy.

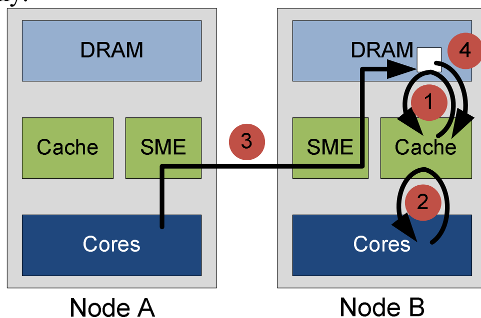

Figure 6. A live wait mechanism that leverages local coherency

In order to leverage cache coherency, the synchronization variables must be mapped as DRAM and not MMIO type, i.e. these variables must reside in local main memory. Then more than one outstanding load to these variables is possible. In order to ensure not only caching but also coherency and consistency, updates from remote nodes must also update (i.e. invalidate) local cache copies. This can be achieved by setting the ‘coherent’ bit in the corresponding HT packets that update the synchronization variables. In this way, synchronization between two threads can be done by having one thread polling on a local memory location for a change, while the other (remote) thread is updating this location by using writes. The principle of this wait mechanism is shown in Figure 6.

This figure describes a scenario where node B is hosting a synchronization variable and node A is updating it. Node B is polling in a coherent manner for a change of this variable. The following steps happen:

1, 2) The thread executed on a core of Node B checks for a change by spinning on a variable. The first access will produce a cache miss and, therefore, result in a memory access. Afterward, it spins on its local cache.

3) A remote thread (on Node A) issues a change that is written-through across the network to node B.

4) The memory controller receives the write operation and triggers the coherency mechanism that invalidates cache copies. The next access of the waiting thread will result again in a cache miss and the most recent value is fetched from memory.

In order to test this idea, two nodes are used: the *DRAM* node hosts the barrier, and the *MMIO* node accesses the barrier on the previous node. The DRAM node will execute the master thread while the MMIO node will host the slave thread. On the *MMIO* node’s side all accesses to the barrier must still traverse the SME.

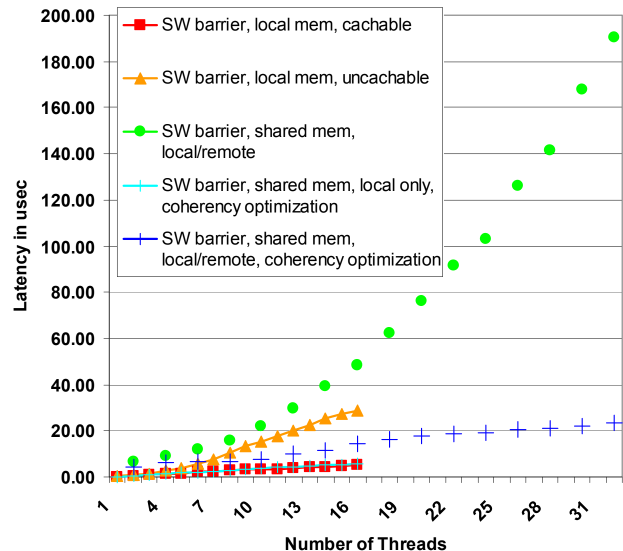

Figure 7. Small scale barrier performance using coherency optimizations

The resulting change in performance is significant: while the master thread polls on the flags of the other threads it can leverage its caches. If an update of a thread from the *MMIO* node comes, this cache copy is invalidated. Of course, the

same applies for an update from a thread running on the *DRAM* node. Similarly, in the opposite direction, a thread on the remote node polling for a change of the phase will always get the most recent version, independently if it originates from main memory (on the *DRAM* node) or from a cache copy (also on the *DRAM* node). As can be seen, this optimization mainly improves master performance

The resulting impact is shown in Figure 7. Here, we compare the execution of up to 32 threads running on two nodes. It can be clearly seen that the optimized version is outperforming the previous versions significantly.

B. *Introducing dissemination and local masters*

Similarly to the optimization done for message passing implementations (see related work), we have further optimized our barrier by disseminating the phase variable into all the local coherency domains (nodes) in the cluster. In this way, every thread can wait for the phase change spinning in local cache as depicted in Figure 7. Therefore, no remote load is needed in this new version of the barrier algorithm and the inter-node traffic becomes constant and minimum. This optimization mainly enhances performance for slave threads.

On the other hand, one thread per coherency domain (node) could be designated as local master. In this way, the global master has to check only one flag per node, thus reducing this sequential process. However, when the number of nodes in the system increases, this optimization may be insufficient as network congestion can be expected if a single gather/scatter point in the cluster is used. Nevertheless, a hierarchical communication mechanism widely discussed in academia [24] can easily solve this problem.

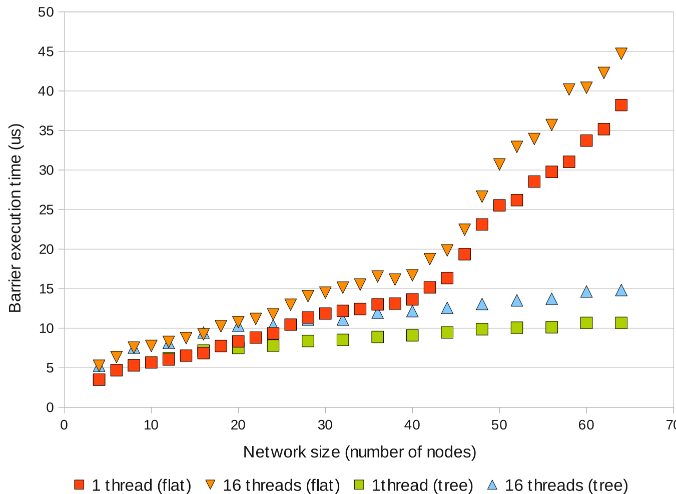

Figure 8. Optimized barrier performance

Figure 8 shows the results for our final barrier implementation. This time the barrier has been extended up to 64 nodes and up to 16 threads per node. Results labeled as *flat* implement a single master in the whole cluster and, thus, the SME at the master gets congested at 40 nodes, as it can be inferred from the clear change on the scalability trend at that number of nodes. To solve this problem we implement the mentioned tree organization to limit the number of notifications a node can receive to only 8. Results labeled as *tree* show the effectiveness of this approach.

By comparing to other related work we can appreciate the outstanding low latency of our barrier: [23] reports a barrier latency of 20us for 8 nodes (only for node counts of a power of two, otherwise 30us), and 8 threads per node, interconnected by Infiniband, while our barrier takes less than 6us for that number of nodes and threads. [22] shows a minimum barrier latency of 38us for 64 nodes and 1 thread per node while our latency for that configuration is 11us. As can be seen, despite using an FPGA implementation, our barrier latency is remarkably low, 15us for 1024 total threads.

VI. Exascale Lookout

Up to now, we have shown that this barrier implementation nicely scales up to 64 nodes and 1024 threads. Results show very regular characteristics, thus allowing performing an extrapolation. Thus, based on our experiments, we are able to estimate the performance for larger systems. In order to do so, two aspects have to be taken into account, (1) the number of threads per node and (2) the number of nodes per system. It is commonly accepted that both values will continue to increase significantly in the future. For the extrapolation, we assume that the barrier latency scales with regard to two input variables:

1. **logarithmically with network size** (i.e. node count), as similar techniques like used in butterfly barriers are employed.
4. **linearly with node size** (i.e. number of cores per node), as the coherency overhead is preventing similar scalability like above. Without coherency, scalability for intra-node communication would also be logarithmic.

A. *Scaling the number of nodes per system*

Based on the measurements from Figure 8, an extrapolation can be set up. Figure 9 predicts the barrier latency depending on node and network size. Each line represents a different node size in terms of amount of cores per node.

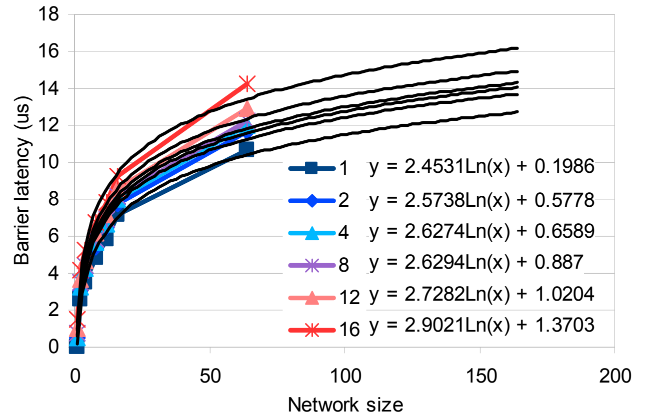

Figure 9. Trend lines for network size

The trend lines' coefficient of determination (R^2) is at least 0.98, which indicates that our assumption on logarithmic scale is correct. For a linear scale, R^2 would drop to about 0.88, showing much worse fitting.

B. Scaling the number of cores per node

Currently we can leverage in our prototype up to 16 cores per node, but we expect that core count in future systems continues increasing. Thus, we also need an extrapolation of barrier latency with regard to node size. As said before, due to the coherency protocol overhead we are expecting a linear growth here[2]. The results from the following figure validate this assumption.

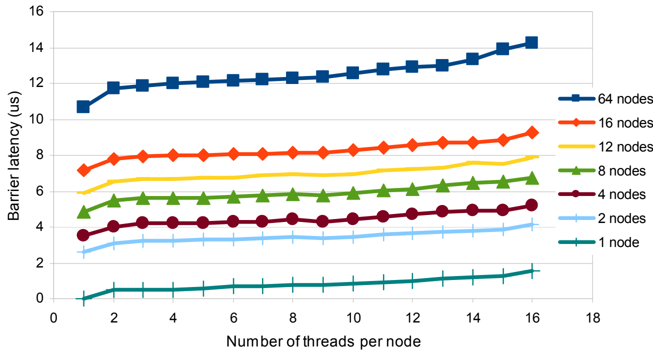

Figure 10. Varying the number of threads per node

For the formulas for network size (Figure 10) a linear best fit is applied to the number of cores/node. I.e. for a formula of the type $y = a \cdot \ln(x) + b$ the coefficients (a, b) are extrapolated. The following formula is derived based on a variable network size (n_{nodes}) and variable node size (n_{cores}). The formula returns the barrier latency (t) in us.

$$t = (0.0216 n_{cores} + 2.4928) \cdot \ln(n_{nodes}) + (0.0561 n_{cores} + 0.377) \quad \text{(Eq. 2)}$$

C. Exascale calculation

With regard to Exascale computing, it is anticipated that approximately 166M cores in 224,000 nodes are required [21] (p. 188). This results in about 740 cores per node. Despite the uncertainty if things will evolve like expected, or if such an implementation is economically, ecologically or practically feasible, we can apply these numbers to demonstrate the feasibility of our barrier.

The results in Figure 12 are surprising, as our estimations show that synchronization in such a massively parallel Exascale system with an advanced technique like the one proposed here is possible in about 282us (for 256k nodes each with 768 cores, i.e. 200M cores in total). In addition, current top-notch installations, like Jaguar for instance, use about 220k cores. For such an installation our calculations show barrier latencies of 28us (16 cores/node or less), 35us (64 cores/node) or 69us (256 cores/node), depending on the ratio between core count and node count.

[2] Note that probe filtering techniques can reduce the number of probes sent out, but only if there are no cache copies. As we expect every core to spin on a synchronization variable, such techniques will not improve performance, thus increasing the accuracy of our estimations.

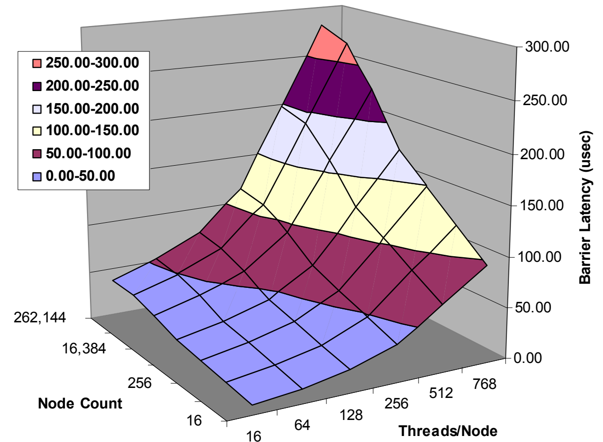

Figure 11. Extrapolation of an exascale barrier latency

VII. INTEGRATING THE BARRIER INTO REAL APPLICATIONS

For MPI environments, leveraging the performance of this barrier technique is as easy as modifying the associated MPI library functions. Instead of doing this, here we propose another opportunity, which is to use shared memory access both for communication and synchronization purposes. In this case, no MPI overhead is present and the system is much easier to program for the user.

Dropping the coherency protocol not only implies a lower memory latency but also some additional advantages, like a complexity reduction in memory controllers and caches, a reduction in the number of required virtual channels in the network (previously devoted to avoid coherency protocol deadlocks), a reduction in the amount of memory used by data related to the coherency protocol (directory and state bits), a reduction in the network traffic, etc. However, the programmer can no longer rely on the coherency mechanism to ensure a sequential consistent framework. In a simplified way, we can suppose that the problem we face is that a copy of a memory block stored in a cache will not be invalidated by a subsequent write operation issued in a different processor and, thus, later read operations can potentially retrieve a stale data from that cache. But, is this an issue a programmer should worry about?

In a system that implements sequential consistency, at the end of the following code, processors 0 and 1 cannot both read 0. However, in our non-coherent system this is likely to happen. Let's assume that variables are initialized to 0:

```
Processor 0:                Processor 1:
1. Store A <- 1             1. Store B <- 1
2. Load B                   2. Load A
```

Although this behavior is not expected by programmers, many current processors do not maintain a sequential consistency indeed. For example, AMD Opteron processors actually allow that both processors read 0 [26]. Note that although the Opteron processor maintains coherency by means of its particular protocol, sequential consistency is broken because of speculative execution of instructions out

of program order. This architectural relaxation allows a faster execution while a sequential consistency can still be achieved with little effort by inserting a memory fence (*MFENCE*) instruction between the store and the subsequent load in the example code. In this way, the Dekker's algorithm for mutual exclusion can be safely implemented. However, a regular programmer does not have to cope with these details because it is common to turn to a mutex library in order to code a mutual exclusive section. Typically, the programmer does not usually implement its own synchronization primitives.

Let us assume a multi-threaded application that uses barriers for synchronization. The code between two barriers performs reads and writes. During a given inter-barrier period, two threads will not write to the same variable because as there is no other synchronization primitive, the variable value would be undetermined. In addition, a thread will not read a variable that is going to be written by another thread in that same inter-barrier period, again because the read value would be undetermined. In this way, we can defer the invalidation process after a write because no other threads will need to read the related variable. As can be seen, the barrier becomes a coherency point in addition to a synchronization point. As a result, the programmer does not need sequential consistency at every single line of code and the restrictions of a typical coherency protocol can be relaxed. To ensure that each thread will read the expected value, it is enough to update the state of the system just at the barriers (in a barrier-based application). For applications based on mutexes it is similar, although these other primitives will not be studied in this paper.

To achieve coherency at a barrier we can have recourse to any of the several mechanisms proposed by papers related to relaxed consistency models [25]. However, as we are using a real prototype we cannot modify processors or memories to implement a given mechanism. On the contrary, we have to leverage the existing possibilities that our circumstantial hardware provides. A straightforward method to change the state of a system from incoherent to coherent is to invalidate every cache in order to avoid multiple copies of a variable in the system. In practice, during the period between the arrival of the last thread to the barrier and the exit of the first thread from that barrier, every cache in the whole system must be invalidated. To do so, we have implemented a simple system call in Linux kernel that uses the *wbinvd* assembler instruction to flush every cache line in the memory hierarchy.

To show how our barrier integrates into a parallel application, we have chosen the fast Fourier transform included in the Splash2 benchmark suite as it uses barriers for synchronization among threads. In this case, we do not focus on barrier performance. This experiment will serve as a demonstrator of our non-coherent system. As we have explained, no modifications to its code are needed. However, certain mallocs will allocate memory in the global memory pool and the barrier calls will invoke our barrier implementation.

Figure 12. FFT execution on global memory (uncachable)

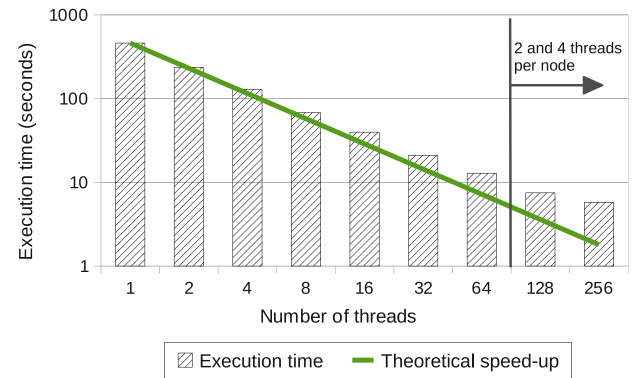

Figure 13. FFT execution on global memory (cachable)

Figures 12 and 13 show the execution time for the tests (correctness of results has been checked). The execution of the FFT application has been distributed across the 64-node cluster. Note that data has not been distributed according to locality. On the contrary, data has been randomly distributed across the global memory pool (a tailored data distribution would improve performance). In this way, an execution with 1 thread located at a single node will still access every node in the cluster.

By comparing both figures, we can see the importance of caching. Additionally, results show an achieved scalability quite near the optimum. Only with 64 nodes the speed-up drops behind the theoretical one due to network congestion. As we said, data allocation has been randomly distributed, thus forcing an all-to-all communication in practice. Thus, a better memory allocation policy would allow better scalability for larger number of nodes. However, note that a system with a coherency protocol would reach congestion much before than ours. As for the current SME configuration, the CPUs constrain the number of outstanding memory requests to one per socket, no performance improvements are obtained beyond 4 threads per node.

This example illustrates how easy it is to execute a common shared-memory application in a cluster while achieving good scalability.

VIII. Conclusion

We have developed a highly scalable barrier for clusters by avoiding limitations of message passing. Instead, we rely on a non-coherent global address space. We show that barrier performance is dramatically improved by architectural optimizations, while technological improvements by using ASICs instead of FPGAs only provide little benefit.

We would like to highlight that best to our knowledge this is the lowest-latency barrier implementation without dedicated networks for synchronization purposes, being about 3.3 times faster than the best most recent implementation [23], which also uses a high performance cluster interconnect, and in spite of an inferior technology (ASIC vs. FPGA) and less powerful CPUs (3.0GHz vs. 2.2GHz).

This barrier can also be applied to message passing applications by only modifying libraries. However, we also show the feasibility of avoiding message passing overhead and easing programming by executing applications using non-coherent distributed shared memory instead of message passing, in particular by executing a shared-memory FFT implementation on our prototype cluster. Experiments with up to 256 threads show the high scalability of this approach.

Future work will include work on other synchronization primitives like locks as well as work on consistency and task models to support this new architecture.

Acknowledgements

This work was supported by the Spanish MEC and MICINN, as well as European Commission FEDER funds, under Grants CSD2006-00046 and TIN2009-14475-C04.

References

[1] Brightwell, R., Riesen, R., and Underwood, K.D. 2005. Analyzing the Impact of Overlap, Offload, and Independent Progress for Message Passing Interface Applications. *Int. Journal of High Performance Computing Application (IJHPCA),* 19(2): 103-117, May 2005.

[2] SGI. *Technical Advances in the SGI Altix UV Architecture*. White Paper. Available at http://www.sgi.com/products/servers/altix/uv/.

[3] ScaleMP. http://www.scalemp.com.

[4] Montaner, H., et al. 2010. Getting Rid of Coherency Overhead for Memory-Hungry Applications. In *IEEE International Conference on Cluster Computing (CLUSTER)*, Sept. 2010, Heraklion, Crete.

[5] Slogsnat, D., Giese A., Nüssle M., and Brüning U. An Open-Source HyperTransport Core. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(3):1-21, 2008.

[6] Nüssle, N., Geib, B., Fröning, H., and Brüning, U. 2009. An FPGA-based custom high performance interconnection network. In *International Conference on Reconfigurable Computing and FPGAs*, Cancun, Mexico.

[7] Litz, H., Fröning, H., and Brüning, U. 2010. HTAX: A Novel Framework for Flexible and High Performance Networks-on-Chip. In *Fourth Workshop on Interconnection Network Architectures: On-Chip, Multi-Chip (INA-OCMC)* at HIPEAC 2010, Jan. 2010, Pisa, Italy.

[8] Fröning, H., and Litz, H. 2010. Efficient Hardware Support for the Partitioned Global Address Space. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW),* Apr. 2010, Atlanta, Georgia.

[9] Fröning, H., Nüssle, N., Slogsnat, D., Litz, H. and Brüning, U. 2006. The HTX-Board: A Rapid Prototyping Station. In *3rd Annual FPGAworld Conference*, Stockholm, Sweden.

[10] Almási, G., et al. 2005. Optimization of MPI collective communication on BlueGene/L systems. In *19th annual international conference on Supercomputing (ICS)*. New York, NY, USA, 253-262.

[11] Mellor-Crummey, J. M., and Scott, M. L. 1991. Algorithms for scalable synchronization on shared-memory multiprocessors. In *ACM Transactions on Computing Systems,* 9(1):21-65.

[12] AMD. *AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions*. Pub. No. 24594, Rev. 3.15, Nov. 2009.

[13] Eugene D. Brooks, III. 1986. The butterfly barrier. *International Journal of Parallel Programming.* 15(4):295-307.

[14] Buntinas, D., Mercier, G., and Gropp, W. 2007. Implementation and evaluation of shared-memory communication and synchronization operations in MPICH2 using the Nemesis communication subsystem. *Parallel Computing,* 33(9):634-644.

[15] Subramoni, H., Kandalla, K., Sur S., and Panda, D. K. 2010. Design and Evaluation of Generalized Collective Communication Primitives with Overlap using ConnectX-2 Offload Engine. In *18th Int'l Symposium on Hot Interconnects (HotI)*, Aug. 2010, Mountain View, California, USA.

[16] Rabenseifner, R. 2004. Optimization of collective reduction operations. In *Intl. Conf. on Computational Science, LNCS 3036*, Springer, 1-9.

[17] Barnett, M., et al. 1993. Global combine on mesh architectureswith wormhole routing. In *7th International Parallel Processing Symposium*, Apr. 1993, Newport, CA, USA.

[18] Panda, D.K. 1995. Global reduction in wormhole k-ary n-cube networks with multidestination exchange worms. In *9th International Symposium on Parallel Processing (IPPS '95)*. IEEE Computer Society, Washington, DC, USA, 652-659.

[19] Gupta S. K. S., and Panda, D. K. 1993. Barrier Synchronization in Distributed-Memory Multiprocessors using Rendezvous Primitives. In *7th Int'l Parallel Processing Symposium (IPPS '93)*.

[20] Hensgen, D., Finkel, R., and Manber, U. 1988. Two algorithms for barrier synchronization. *International Journal of Parallel Programming,* 17(1):1-17, Feb. 1988.

[21] Kogge P. (Ed.), et al. 2008. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. US Department of Energy, Office of Science, Advanced Scientific Computing Research, Washington, DC. Available at http://www.er.doe.gov/ascr.

[22] Hoefler, T., et al. 2006. Fast Barrier Synchronization for InfiniBand. In *20th International Parallel and Distributed Processing Symposium (IPDPS)*, Apr. 2006, Rhodes Island, USA.

[23] Graham, R.L., et al. 2010. Overlapping computation and communication: Barrier algorithms and ConnectX-2 CORE-Direct capabilities. In *2010 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum (IPDPSW),* Apr. 2010, Atlanta, GA, USA.

[24] Hoefler, T., Mehlan T., Mietke F., and Rehm, W. 2004. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. *Chemnitzer Informatik Berichte. Vol 04, Nr. 03*, ISSN: 0947-5152, Dec. 2004.

[25] Adve, S.V., and Gharachorloo, K. 1996. Shared memory consistency models: a tutorial. *Computer,* 29(12):66-76, Dec 1996.

[26] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*. Pub. No. 24593, Rev. 3.17, June 2010.

[27] Hoffmann, H., Wentzlaff, D., and Agarwal, A.2010. Remote Store Programming: A Memory Model for Embedded Multicore. In *Lecture Notes in Computer Science*, Volume 5952/2010, 3-17.