Towards Efficient MapReduce Using MPI

Torsten Hoefler[1], Andrew Lumsdaine[1], and Jack Dongarra[2]

[1] Indiana University, Open Systems Lab, Bloomington, IN, USA
{htor,lums}@cs.indiana.edu
[2] Department of Computer Science, University of Tennessee Knoxville
dongarra@eecs.utk.edu

Abstract. MapReduce is an emerging programming paradigm for data-parallel applications. We discuss common strategies to implement a MapReduce runtime and propose an optimized implementation on top of MPI. Our implementation combines redistribution and reduce and moves them *into the network*. This approach especially benefits applications with a limited number of output keys in the map phase. We also show how anticipated MPI-2.2 and MPI-3 features, such as MPI_Reduce_local and nonblocking collective operations, can be used to implement and optimize MapReduce with a performance improvement of up to 25% on 127 cluster nodes. Finally, we discuss additional features that would enable MPI to more efficiently support all MapReduce applications.

1 Introduction

MapReduce [1,2] is an emerging programming framework to express data-parallel applications and algorithms. After its original cluster-based implementation, it has been implemented and evaluated on various architectures, including the Cell B.E. [3], GPUs [4], and multi-core processors [5]. Given the large problem sizes that are addressed using MapReduce, and given the popularity of MapReduce as an implementation paradigm, it seems natural to explore its use on traditional HPC platforms. Accordingly, in this work we discuss implementation and optimization of MapReduce functionality using MPI. Based on our experiences, we discuss several extensions to MPI that would enable more natural support of MapReduce.

The key functionality in MapReduce, i.e., **Map** and **Reduce**, are analogous to the functional programming constructs *map* and *fold*. These functional constructs are found in many modern programming languages. For example, C++'s Standard Library offers `std::transform()` and `std::accumulate()` that respectively provide functionality similar to map and fold.

The MapReduce model defines a two-step execution scheme that can be effectively parallelized. The user only defines two functions, $\mathcal{M} : (K_m \times V_m) \mapsto (K_r \times V_r)$ (map) which accepts input key-value pairs $(k, v)\, k \in K_m,\, v \in V_m$ and emits output key-value pairs $(g, w)\, g \in K_r,\, w \in V_r$, and a function $\mathcal{R} : (K_r, V_r^{\mathbb{N}}) \mapsto (K_r, V_r)$ (reduce) which accepts a key $g \in K_r$ and a list of values $v \in V_r^{\mathbb{N}}$ and generates a single return value. The MapReduce framework accepts a list of input values $(K_m \times V_m)^N$, $N \in \mathbb{N}$, calls $\mathcal{M}$ for each of the N inputs and collects the *emitted* result pairs. Then it groups all result pairs by their key g and calls $\mathcal{R}$ for each key and the associated list of values. The functions $\mathcal{M}$ and $\mathcal{R}$ are defined to be pure functions (without side effects), i.e., they solely compute the output based on immutable inputs (without internal state). This means that the order of application of $\mathcal{M}$ (and $\mathcal{R}$ for different g) is independent and can be executed in parallel. Ordering constraints are only imposed by data-dependencies (i.e., all data must be produced before it can be consumed), such that no synchronization is necessary. However, we note that in the general case where any map task can emit any key $g \in K_r$, an implicit barrier exists between the map and reduce tasks. This will be discussed in Section 2.

Many data-parallel algorithms can be expressed in this framework. Prominent examples are sorting, counting elements in lists (e.g., words in documents), distributed search (grep), or transposition of graphs or lists [1]. More complex algorithms, such as Bellman Ford single source shortest path or PageRank [6] can be modeled by iteratively invoking MapReduce. It has also been shown that MapReduce can be applied in the context of advanced research in machine learning [7]. The purity of the two functions allows the runtime to perform several optimizations. The most obvious strategy is the concurrent execution of map and reduce tasks in the two phases. This efficient auto-parallelization has been proposed in [1]. Another very useful characteristic of MapReduce in large-scale systems is its inherent fault resiliency because map or reduce tasks on failed or slow nodes can simply be restarted on other nodes. Thus, MapReduce allows the application developer to focus on the important algorithmic aspects of his problem while allowing him to ignore issues like data distribution, synchronization, parallel execution, fault tolerance, and monitoring. Thus, MapReduce also gained significant popularity in education [8].

2 MapReduce Communication Requirements

We start by discussing the communication requirements of MapReduce and possible implementations. Figure 1 shows the two phases of MapReduce and the necessary communication (data dependencies). Data dependencies are:

a **reading of input for $\mathcal{M}$:** the map tasks need to read (possibly large amounts of) input data of size $\Omega(N)$ (with N input pairs)
b **building input lists for $\mathcal{R}$:** all pairs need to be ordered by keys $g \in K_r$ and the lists of total size $\mathcal{O}(N)$ need to be transferred to the reduce tasks
c **output data of $\mathcal{R}$:** the reduce tasks output the data; this is usually a negligible operation of $\mathcal{O}(|K_r|)$ if we assume the common case $|K_r| \ll N$

Thus, the two critical data movements are collecting the input for the map tasks and arranging the output of the map tasks as input for the reduce tasks.

Required data movement obviously depends on the parallelization strategy. For example, if all map and reduce tasks are executed on a single node, then no data movement is necessary. However, parallelism is the most important feature of the

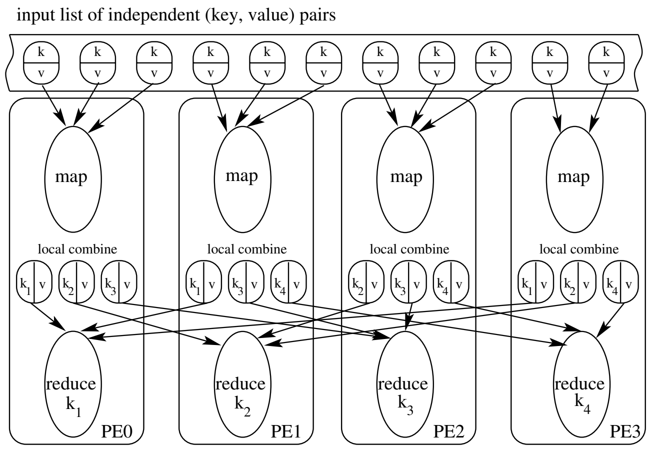


Fig. 1. MapReduce Communication Scheme

MapReduce model. The maximum parallelism of the embarrassingly parallel map phase is only limited by the number of input pairs N. Ad extremum, each application of $\mathcal{M}$ is done as a separate map task on a separate processing element (PE). However, if P PEs are available, it is usually not advisable to process each reduce separately because the overhead of starting and administering tasks can be very high compared to a single application of $\mathcal{M}$ and often $N \gg P$ holds. Thus, the input is commonly divided in reasonably sized work packets (e.g., 64 MiB in [1,9]) that are processed in parallel on all available PEs. A manager process can be used to administer the distribution of those packets and ensure proper actions in cause of node-failures (e.g., resubmit the map tasks to other nodes). Multiple different strategies can be defined for reading the input data for each map task. The most common case is that a shared filesystem is available and the tasks just read the data directly from (local) disk. The manager tries to assign map tasks to nodes that are close to the needed data. However, mapping tasks close to data might not be possible in certain environments so that the map tasks need to collect the data before they can start. In the extreme case, only the manager process has access to the data and needs to distribute it to the workers.

The parallelism in the reduction phase is limited by the number of different output keys of the map phase ($|K_r|$) which highly depends on the implemented algorithm *and* the input data. Additionally, each application of $\mathcal{R}$ needs *all* results of the map phase with the same key. Depending on the key distribution in the input pairs, this can effectively be an (irregular) all-to-all exchange with $P \times |K_r| \in \mathcal{O}(|K_r|^2)$ messages. It can again be tried to minimize the data movement with appropriate placement of the reduce tasks (close to their keys [10, 11]), however, this heavily depends on the input set and many applications do not seem to exhibit such a regularity (cf. parallel sort or string search). Thus, we have to assume the worst case, a synchronizing all-to-all exchange.

Throughout the remainder of the paper, we assume the hardest case where the input pairs are only available on the master process and the keys in K_r are evenly distributed among all items in the input data. We note that MapReduce is often used to transform very large datasets, where the map tasks need to be moved to the data. However, the redistribution before the reduce phase can not easily be optimized for the general application in this way.

2.1 A Simple MPI Implementation

A straight-forward MPI implementation which is very similar to existing frameworks such as Hadoop [9] or Google's implementation [1] can be easily implemented with MPI point-to-point communication. Each idle worker process queries the master for work packets. In the first stage, the master assigns map tasks to the processing elements. After all map tasks are done, the master disseminates reduce tasks. Each reduce task queries all available PEs for the values associated with its key. This method seems rather unfavorable because it tends to create many hot spots in the network (e.g., at the master process) and communication between map and reduce can easily be unbalanced potentially resulting in a severe performance degradation.

MPI has several mechanisms to optimize parallel computations. Two common optimization possibilities are (1) collective operations, which apply intelligent communication patterns to reduce congestion and minimize the number of message transmissions, and (2) overlapping communication and computation. Thus, a possible improvement would be to assign all reduce tasks to PEs before the second data exchange is started and perform it as a collective MPI_Alltoallv operation. While this has some potential to reduce network congestion, it is very tricky to optimize the general MPI_Alltoallv operation and it is generally accepted to be the least-scalable collective operation in MPI (due to the high number of unpredictable message exchanges $\mathcal{O}(P^2)$). Overlapping communication and computation can be used up to a certain extent if the master sends the input data for the map tasks in a pipelined fashion. However, it seems unreasonable to apply such techniques to the redistribution phase because reduce processes can only start when *all* input data is available.

This trivial implementation does not fully utilize all possible features of high-performance computing (HPC) systems. Thus, in the following section, we propose an orthogonal approach in order to take advantage of more scalable MPI functions and advanced optimization techniques.

3 Scalable MapReduce in MPI

In this section, we describe a different HPC-centric approach for the solution of the redistribution problem. We use a typical MapReduce program, the search for the number of given strings in a file, as example. The map function (Listing 1.1) accepts an input file (or a part of it) and a vector of strings s. It searches the input and emits the pair $(s,1)$ for each occurrence of string s. The reduce function (Listing 1.2), which is also used as local combiner, simply counts the number of elements for a given s.

```
void map(filem f, keys strs) {
 for(i=0; i<strs.size(); i++) {
  char *ptr=f.start_addr();
  while(ptr<f.end_addr()-strs.len) {
   if(!memcmp(ptr, str[i], strs.len))
    EmitIntermediate(i, 1);
   ptr++;
 } } }
```

Listing 1.1. Map Function

```
void reduce(key str, values num) {
  int sum=0;
  for(i=0; i<values.size(); i++) {
    sum += values[i];
  }

  Emit(sum);
}
```

Listing 1.2. Reduce Function

As discussed before, MapReduce readily lends itself to be implemented in a master-worker model, i.e., the computation (map and reduce tasks) can be scheduled by a central master process and executed by worker processes. MPI is well suited to implement this concept by using rank 0 as master process and all other $P-1$ ranks as workers. We now discuss how to use collective operations to perform the map and reduce task. The map task can be implemented as a simple MPI_Scatter operation and the reduce task can use an MPI_Reduce operation. This basically moves the reduction function $\mathcal{R}$ into the MPI library (i.e., the network layer). Either built-in reductions or MPI user-defined reduction operations, as described in [12] Section 5.9.5, can be used as $\mathcal{R}$. The implementation as user-defined reduction puts several limitations on $\mathcal{R}$ and the input data:

I $\mathcal{R}$ must be associative (MPI reduction operations are generally assumed to be associative)
II the number of different keys $|K_r|$ must be known by each process (or communicated in advance, e.g., with MPI_Allreduce) and, the values of all keys $g \in K_r$ must be fixed-size elements (can be arbitrary MPI datatypes). The output lists for each key can be reduced locally with a *combiner* [1]. MPI-2.2 will very likely include MPI_Reduce_local which can be used to implement the combiner trivially.
III if not all processes have a value for each key $g \in K_r$, then an identity element with respect to $\mathcal{R}$ has to exist and must be supplied by the processes without values.

Using scalable collective MPI reductions has a high optimization potential. It enables the application to take advantage of optimized implementations of collective operations. Several architectures, such as BlueGene [13], or networks, such as Quadrics [14], support hardware-optimized collective operations (they even perform some simple reduction operations directly in the network hardware). This implementation inherently supports the problematic case where the number of keys is small (the parallelism in the reduce phase is limited). In this case, tree-based algorithms can be used to reduce the number of messages from $\mathcal{O}(|K_r|^2)$ (cf. Section 2) down to $\mathcal{O}(log_2(|K_r|))$ on fully connected networks. However, if many keys need to be reduced, the transmission time is likely going to be $\mathcal{O}(|K_r|)$ due to bandwidth limitations at the master node. The adaptation to different network topologies can be done with specialized collective operations. The reduction in our example fulfills all requirements and can be implemented as a simple MPI_SUM reduction operation. We note that this approach to implement $\mathcal{R}$ is fundamentally different to previous approaches. The reduction operations are not applied in parallel, but the data is reduced during the communication itself. Obviously, local memory and the size of each value limits the maximum number of keys. If this is an issue, the MapReduce framework could just fall back to the all-to-all-based parallel reduction approach or perform the reduce phase in multiple MPI_Reduce steps.

3.1 A Simple MapReduce Example

The string search example shows the applicability of the MPI-based MapReduce scheme well. However, we decided to implement a more flexible application that is able to simulate a large class of MapReduce programs. This application accepts four parameters: the number of tasks, minimum and maximum task duration t_{min}, t_{max}, and the size of the reduction operation s_{red}. The application issues work-packets that take a random time in the closed interval $[t_{min}, t_{max}]$. The times are uniformly distributed. A worker process retrieves a work-packet and simulates the map function $\mathcal{M}$ by computing for the duration of time indicated in the packet. After the computation is finished, the worker starts the parallel reduction $\mathcal{R}$=MPI_SUM of size s_{red} (by calling MPI_Reduce). The execution scheme is shown in Figure 2(a).

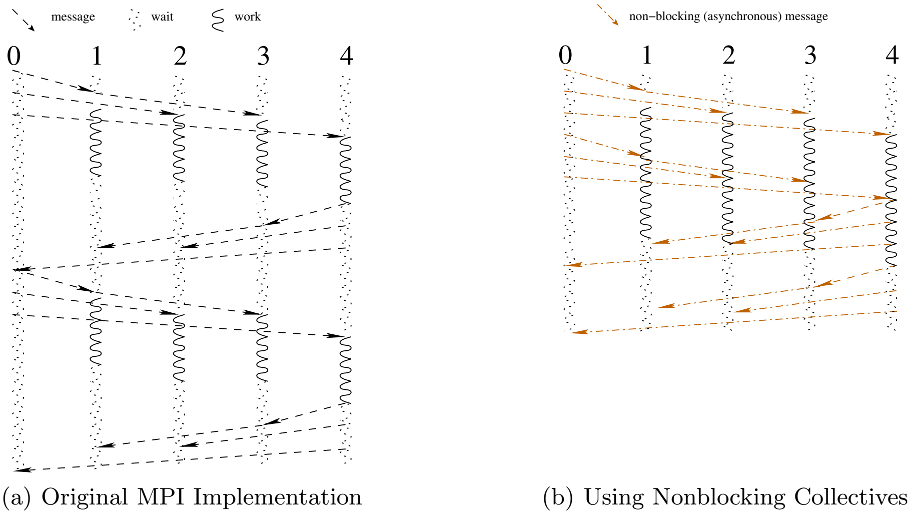

message
wait
work
0
1
2
3
4
(a) Original MPI Implementation
non–blocking (asynchronous) message
0
1
2
3
4
(b) Using Nonblocking Collectives

Fig. 2. HPC-centric MapReduce Scheme assuming 5 processes executing 8 tasks and a binomial tree scatter and reduction algorithm

3.2 Further Optimization Possibilities

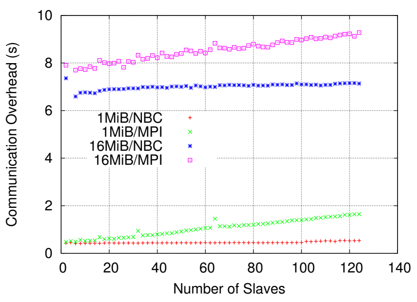
Common optimizations for parallel programs are optimizing communication patterns with collective operations, which we have already done, and hiding latency by overlapping communication and computation. We discussed in Section 2.1 that overlapping can be done with pipelining data in the map phase but that the redistribution phase can not be transformed easily. In our optimized scheme, where the data redistribution and the reduce phase are merged, a similar pipelining scheme can be utilized. However, to retain the benefits of optimized collective operations, we would need a nonblocking version of those operations to enable overlap. Such nonblocking collective operations are proposed for MPI-3 [15] and a reference implementation exists with LibNBC [16].

We can easily apply nonblocking collective operations to the map as well as the reduce functionality. In the map-case, the master starts w_m nonblocking NBC_Iscatter operations at the beginning. Then, it waits for the first one to complete and starts the next operation after one completed. This efficiently creates a *window* of scatter operations that run in the background. Their latency can be ignored if the window size is large enough and the work-packets take long enough to compute. The trade-off there is that the memory requirements grow with the window size, i.e., data for all running operations has to be in main memory at the root process. The worker processes similarly start w_m non-blocking scatter operations and re-post new non-blocking scatter until the signal to exit is received. In the reduce case, we have to define a set of w_r buffers to support w_r outstanding nonblocking operations. A similar window-technique as in the map operation is used to have w_r outstanding NBC_Ireduce operations at any time. If all n buffers are in use, the *oldest* reduction has to be finished by calling NBC_Wait on the master process and re-posting a new NBC_Ireduce. The remaining outstanding communications have to be finished and their buffers reduced when all tasks are completed. The resulting execution scheme is shown in Figure 2(b).

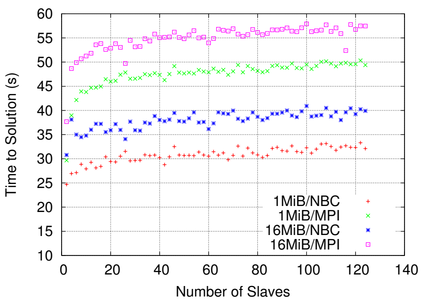
3.3 Performance Results

MapReduce applications typically process large amounts of data that have to be read from either the network or local disks. Thus, we assume that the I/O bandwidth is not sufficient to keep multiple processing elements busy. However, most of today's systems are multi-core or SMP systems such that there are idle cores available to offload the communication. We use the threaded InfiniBand-optimized version of LibNBC [17,18] for all benchmarks. This efficiently results in offloading the reduce task to another core (the reduce operation is a part of the NBC_Reduce communication) and thus utilizes another level of functional parallelism transparently to the application developer. Benchmarks of the simple string-search example were also covered by the more extensive simulator and delivered exactly the same results. Thus, we only present benchmark results for the different configurations of the simulator.

We benchmarked two different workload-scenarios with 1 to 126 worker nodes with 10 tasks per process. We compared the threaded version of LibNBC with a maximum of 5 outstanding collective operations with Open MPI 1.2.6. We also varied the data-size of the reduction operation (in our example, we used MPI_SUM as the reduction operation).



(a) Communication Overhead of Static Workload



(b) Time to Solution of Dynamic Workload

Fig. 3. Overhead and Time to Solution for Static and Dynamic Workloads for different Number of Workers

Figure 3(a) shows the communication and synchronization overhead for a static workload of 1 second per packet. Using nonblocking collective results in a significant performance increase because nearly all communication can be overlapped. The remaining communication overhead is due to InfiniBand's memory registration which is done on the host CPU. The graphs show a reduction of communication and synchronization overhead of up to 27%. Figure 3(b) shows the influence of nonblocking collectives to dynamic workloads varying between $1\,ms$ and $10\,s$. The significant performance increase is due to avoidance of synchronization and the use of communication/computation overlap. This clearly shows that our technique can be used to benefit MapReduce-like applications significantly. The dynamic example shows improvements in time to solution of up to 25% over the unoptimized implementation.

4 What Is MPI Missing?

MapReduce was not intended to be implemented on top of MPI, and MPI lacks several features that are needed to implement it efficiently.

One of the most important features of MapReduce is its ability to handle faults transparently. However, the default error handling in MPI (MPI_ERRORS_ARE_FATAL) is to abort the job. The user can change this default to return an error from the failed function (MPI_ERRORS_RETURN). While this failure mode might help for point-to-point communications, collective communication can not easily recover from such an error state. Moreover, checking if a collective operation completed successfully on all nodes is not trivial (or very expensive). One alternative is the use of intercommunicators for point-to-point communications [19], but again, collective communication is not handled. An efficient and fault-tolerant implementation would require extended fault tolerance support in MPI. This does not necessarily mean changes in the API [19], but it should support rebuilding of communicators and (partial) restarting of collective communications. First results in this direction have been demonstrated by FT-MPI [20].

A second barrier in the usage of MPI for general MapReduce algorithms are the restrictions of the intermediate data and of the function $\mathcal{R}$ as discussed in Section 3. In order to support arbitrary MapReduce operations, MPI would need variable-sized reduction operations because $\mathcal{R}$ can return values that are of a different size than the input values, e.g., string concatenations. Such a feature would also be desired in the support of higher languages, such as C# [21].

Another feature that can be used to optimize the implementation as shown in Section 3.2 are nonblocking collective operations [15]. A proposal for this class of operations is discussed by the MPI Forum for inclusion in MPI-3.

5 Conclusions and Future Work

MapReduce and MPI were developed in two different communities that have traditionally been somewhat disjoint. However, as the needs and capabilities of these two communities continue to converge, it will be to the benefit of both to leverage their respective technologies. In the case of MapReduce and MPI, we have seen that is possible to efficiently implement MapReduce using MPI – with some limitations. For example, HPC-centric optimizations can be applied if the reduce function fulfills certain criteria. Additional performance gains are possible through upcoming MPI features. Using nonblocking collective operations, for example, provided a speedup of up to 25% over the blocking implementation.

Fully supporting MapReduce will require several additional features and capabilities from MPI. However, many of these features are generally recognized as being important, particularly as MPI evolves to support other modern programming and parallelization paradigms.

Acknowledgments

This work is supported by a grant from the Department of Energy project FastOS II (LAB 07-23) and a grant from the Lilly Endowment.

References

1. Dean, J., Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters. Commun. ACM 51, 107–113 (2008)
2. Lämmel, R.: Google's MapReduce programming model – Revisited. Sci. Comput. Program. 68, 208–237 (2007)
3. de Kruijf, M., Sankaralingam, K.: MapReduce for the CELL B.E. Architecture. IBM Journal of Research and Development 52 (2007)
4. He, B., Fang, W., Luo, Q., Govindaraju, N.K., Wang, T.: Mars: a MapReduce framework on graphics processors. In: PACT 2008: Proceedings of the 17th international conference on Parallel architectures and compilation techniques, pp. 260–269. ACM, New York (2008)

5. Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., Kozyrakis, C.: Evaluating MapReduce for Multi-core andMultiprocessor Systems. In: HPCA 2007: Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture, Washington, DC, USA, pp. 13–24. IEEE Computer Society, Los Alamitos (2007)
6. Langville, A.N., Meyer, C.D.: Google's PageRank and Beyond: The Science of Search Engine Rankings. Princeton University Press, Princeton (2006)
7. Chu, C.T., Kim, S.K., Lin, Y.A., Yu, Y., Bradski, G.R., Ng, A.Y., Olukotun, K.: Map-Reduce for Machine Learning on Multicore. In: Schölkopf, B., Platt, J.C., Hoffman, T. (eds.) NIPS, pp. 281–288. MIT Press, Cambridge (2006)
8. Kimball, A., Michels-Slettvet, S., Bisciglia, C.: Cluster computing for web-scale data processing. SIGCSE Bull. 40, 116–120 (2008)
9. Hadoop (2009), `http://hadoop.apache.org`
10. Pike, R., Dorward, S., Griesemer, R., Quinlan, S.: Interpreting the data: Parallel analysis with Sawzall. Scientific Programming 13, 277–298 (2005)
11. Ghemawat, S., Gobioff, H., Leung, S.T.: The Google file system. SIGOPS Oper. Syst. Rev. 37, 29–43 (2003)
12. Message Passing Interface Forum: MPI: A Message Passing Interface Standard, Version 2.1 (2008)
13. Gara, A., et al.: Overview of the Blue Gene/L system architecture. IBM Journal of Research and Development 49, 195–213 (2005)
14. Petrini, F., Frachtenberg, E., Hoisie, A., Coll, S.: Performance Evaluation of the Quadrics Interconnection Network. Journal of Cluster Computing 6 (2003)
15. Hoefler, T., Kambadur, P., Graham, R.L., Shipman, G., Lumsdaine, A.: A Case for Standard Non-Blocking Collective Operations. In: Cappello, F., Herault, T., Dongarra, J. (eds.) PVM/MPI 2007. LNCS, vol. 4757, pp. 125–134. Springer, Heidelberg (2007)
16. Hoefler, T., Lumsdaine, A., Rehm, W.: Implementation and Performance Analysis of Non-Blocking Collective Operations for MPI. In: Proceedings of the 2007 International Conference on High Performance Computing, Networking, Storage and Analysis, SC 2007. IEEE Computer Society/ACM (2007)
17. Hoefler, T., Lumsdaine, A.: Optimizing non-blocking Collective Operations for InfiniBand. In: Proceedings of the 22nd IEEE International Parallel & Distributed Processing Symposium, IPDPS (2008)
18. Hoefler, T., Lumsdaine, A.: Message Progression in Parallel Computing - To Thread or not to Thread? In: Proceedings of the 2008 IEEE International Conference on Cluster Computing. IEEE Computer Society Press, Los Alamitos (2008)
19. Gropp, W., Lusk, E.: Fault Tolerance in MPI Programs. Special issue of the Journal High Performance Computing Applications (IJHPCA) 18, 363–372 (2002)
20. Fagg, G.E., Angskun, T., Bosilca, G., Pjesivac-Grbovic, J., Dongarra, J.: Scalable Fault Tolerant MPI: Extending the Recovery Algorithm. In: Di Martino, B., Kranzlmüller, D., Dongarra, J. (eds.) EuroPVM/MPI 2005. LNCS, vol. 3666, pp. 67–75. Springer, Heidelberg (2005)
21. Gregor, D., Lumsdaine, A.: Design and implementation of a high-performance MPI for C# and the common language infrastructure. In: PPoPP 2008: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, pp. 133–142. ACM Press, New York (2008)