Efficient MPI Implementation of a Parallel, Stable Merge Algorithm

Christian Siebert[1] and Jesper Larsson Träff[2]

[1] RWTH Aachen University and German Research School
for Simulation Sciences GmbH
Schinkelstrasse 2a, 52062 Aachen, Germany
c.siebert@grs-sim.de

[2] Vienna University of Technology, Institute of Information Systems
Research Group Parallel Computing
Favoritenstrasse 16, 1040 Wien, Austria
traff@par.tuwien.ac.at

Abstract. We study different approaches to implement an optimal, stable two-way merge algorithm for distributed-memory parallel architectures. The algorithm takes as input two ordered sequences, which are distributed blockwise across all available processes such that each process owns a block of elements of each sequence. The task for each process is to produce an ordered block of elements from the stable merge of the input sequences. We present an optimal, perfectly load-balanced, stable parallel algorithm that accomplishes this task. We describe three different implementation alternatives using one-sided communication of the Message-Passing Interface (MPI). Further, we discuss problematic issues with the current MPI 2.2 one-sided interface and enabling features that may be found in future versions of the MPI standard. Experimental results on a large IBM Blue Gene/P supercomputer show perfect scalability of our implementation: with a fixed input size per process the running time remains (almost) constant with increasing number of processes, and with a fixed total problem size our implementation improves the time to solution for up to 32,768 MPI processes.

1 Introduction

Merging of ordered sequences is a fundamental operation in many applications and a key ingredient for many parallel, notably sorting algorithms. As such it has been studied intensively. However, most parallel merging algorithms are designed for shared-memory architectures [1,2,3,5,6,10,11], and only few algorithms have been described [4] and fewer implemented for distributed-memory architectures. For instance, the latter BSP algorithm builds on shared-memory algorithms that are both unnecessarily complicated and potentially inefficient, in terms of both non-dominant splitting overhead and achieved load balance. Although this algorithm could be implemented in MPI, we are not aware of any such implementation. In this paper we describe an algorithm that is both simpler to implement and better in terms of load balance and overhead. Although

Published in "Proc. of EuroMPI 2012", LNCS 7490, pp. 204–213, Springer, 2012.
The original publication is available at www.springerlink.com

similar in nature to the algorithm presented in [2], our algorithm was discovered independently. A specific improvement is that our algorithm is stable, a desirable property for inputs with duplicate elements. The algorithm employs a logarithmic time preprocessing step, very similar to binary search, which can be naturally expressed with one-sided communication. The specific contribution for the MPI community is that we analyze and experimentally compare implementation alternatives with the one-sided communication model of MPI 2.2 [7]; and show that some of the resulting problems can be resolved with the one-sided model proposed for the upcoming MPI 3.0 standard (see `www.mpi-forum.org`).

2 Distributed, Stable Two-Way Merging

Let $\mathsf{stable\_merge}(A, B)$ denote the stable merge of two ordered arrays A and B. Stability means that elements of A that are equal to elements of B are placed before the elements of B in the output, and that the relative order of any sequence of equal elements in either A or B is preserved in the output. The distributed, stable merging problem is the following. The two ordered arrays A and B with m and n elements, respectively, are distributed blockwise across the available p processes, such that process r for $0 \leq r < p$ has a block of m_r consecutive elements $A[s_r^A, \ldots, s_r^A + m_r - 1]$ and a block of n_r consecutive elements $B[s_r^B, \ldots, s_r^B + n_r - 1]$ with start indices $s_r^A = \sum_{i=0}^{r-1} m_i$ and $s_r^B = \sum_{i=0}^{r-1} n_i$, respectively. Each process r produces a block $C[s_r^A + s_r^B, \ldots, s_r^A + s_r^B + m_r + n_r - 1]$ of consecutive elements of $C = \mathsf{stable\_merge}(A, B)$.

All parallel merge algorithms divide the input sequences into smaller, disjoint, consecutive sequences, that can be merged pairwise in parallel into the corresponding positions of the output array. Our algorithm accomplishes this by using the following idea: given an index i (say the start index $s_r^A + s_r^B$ in C for process r) in the output array C, determine the two indices j and k in the input arrays A and B, such that stably merging the prefixes $A[0, \ldots, j-1]$ and $B[0, \ldots, k-1]$ will produce exactly the prefix $C[0, \ldots, i-1]$ of the stably merged result $C = \mathsf{stable\_merge}(A, B)$. We call j and k the *co-ranks* of i. Put differently, j and k index the first elements of A and B that are *not* among the first i elements of the stably merged output C. For any process r the co-ranks of the start indices $i_r = s_r^A + s_r^B$ and $i_{r+1} = s_{r+1}^A + s_{r+1}^B$ will determine exactly the blocks of A and B needed to produce the output sequence of C for process r. Based on these co-ranks, process r can also determine from which processes to get the input blocks, and perform a local merge on them to produce the final result. All that is needed is an efficient algorithm for computing the co-ranks for any given index i in C. We present and discuss such an algorithm below.

The sequential co-ranking algorithm is given as a C program fragment in Figure 1. It maintains the invariant that $i = j + k$. For both j and k indices lower bound indices are also maintained. For any given input index i with $0 \leq$ `i` $< m + n$ it chooses the largest possible j index in A, and starts out with the assumption that $A[j-1] \leq B[k]$, meaning that all elements of A up to j have to come before the B elements in $\mathsf{stable\_merge}(A, B)$. If this is not the case, the

```
// initialize start indices, invariant i = j+k, j as large as possible
j = min(i,m); k = i-j; j_low = max(0,i-n); // k_low set in first iteration
active = 1;
do {
   // converge indices to the co-ranks
   if (j>0&&k<n&&A[j-1]>B[k]) {
      delta = (1+j-j_low)/2;
      k_low = k;
      j -= delta; k += delta;
   } else if (k>0&&j<m&&B[k-1]>=A[j]) {
      delta = (1+k-k_low)/2;
      j_low = j;
      k -= delta; j += delta;
   } else active = 0; // co-ranks found
} while (active);
```

Fig. 1. Algorithm to find the co-ranks `j` and `k` for an index `i` in the output array C.

index j in A is decreased by halving the interval between j and its lower index `j_low`. Should it turn out that $B[k-1] \geq A[j]$ then instead the k index in B is decreased. To maintain the invariant, whenever either index is halved, the other index is increased by the same amount. The lower bound indices are chosen such that the array bounds m and n cannot be exceeded when an index is increased. Note that the lower index `k_low` for k does not need to be initialized separately, since at the beginning only the first `if` condition may be true, which will cause this index to be initialized properly.

Analysis shows that the algorithm takes at most $\lceil \log_2(\min(m,n)) \rceil + 1$ iterations, since the value `delta` is halved in each iteration, regardless of which branch is taken, and `delta` is initially at most $\min(m,n)$. For brevity, we omit the proof that the co-ranks indeed correspond to the indices for the prefixes needed to produce a stable merge here, although it is not difficult to see.

The distributed version of the algorithm has the input arrays A and B distributed over all processes. The accesses to the array fields therefore potentially entail remote accesses to the memory of other processes. The fully distributed, stable merge algorithm for each process r can be stated as follows:

1. Let $i_r = s^A_r + s^B_r$ be the start index for process r in the output array C. Compute the co-ranks j_r and k_r via a distributed version of Algorithm 1.
2. Get the co-ranks j_{r+1} and k_{r+1} from process $r+1$ (the last process $r = p-1$ sets $j_{r+1} = j_r + m_r = m$ and $k_{r+1} = k_r + n_r = n$).
3. Get $A[j_r \ldots j_{r+1}-1]$ and $B[k_r \ldots k_{r+1}-1]$ from the processes that own these array blocks via communication.
4. Locally compute $\mathsf{stable\_merge}(A[j_r \ldots j_{r+1}-1], B[k_r \ldots k_{r+1}-1])$ to produce the final result $C[i_r \ldots i_{r+1}-1]$.

Theorem 1. *Let m_r+n_r be the maximum number of elements for some process. The above algorithm merges two sequences in time $\mathcal{O}(\log(\min(m,n))+m_r+n_r)$.*

Proof. We assume the co-ranking algorithm used in Step 1 is correct. It completes in $\mathcal{O}(\log(\min(m,n))$ iterations with at most 4 single-element remote memory accesses per iteration. Step 2 requires a communication of only two values. The data exchange in step 3 communicates a total volume of $m_k + n_k$ elements. With a balanced distribution of the arrays, each array block $A[j_r, \ldots, j_{r+1} - 1]$ spans a constant number of processes, so getting the block takes a constant number of communication steps with a total volume of $m_k + n_k$ elements. The local stable merge in Step 4 takes $\mathcal{O}(m_k + n_k)$ operations. In total, our distributed merge algorithm therefore completes in time $\mathcal{O}(\log(\min(m,n)) + m_r + n_r)$. $\square$

The proof assumes that any concurrent read accesses that may occur during Step 1 and Step 3 can be handled efficiently.

3 Implementation Alternatives

The distributed merge algorithm has a straight-forward implementation with any communication interface that supports one-sided communication. Indeed, the MPI 2.2 one-sided communication model [7, Chapter 11] should in principle enable an efficient, highly portable implementation. It offers different implementation alternatives, which we evaluate for our algorithm. The algorithm consists of two main communication phases: first, the co-ranking algorithm requires $\mathcal{O}(\log(\min(m,n)))$ potentially remote single-word accesses per process. The binary search like pattern is data dependent, therefore irregular, and in each iteration only the source process knows which data elements to assess on which processes. This phase is thus a paradigmatic case for one-sided communication. We note that a standard binary search follows much the same pattern, which makes our implementation alternatives also relevant for distributed binary searches in general. During the other main communication phase in Step 3, each process copies the array blocks needed for the local merge. This step can also be expressed conveniently with one-sided communication.

For the implementation alternatives that use the MPI 2.2 one-sided model, we assume that the input arrays A and B are exposed in (two disjoint) *communication windows.* The alternatives differ in how accesses to the windows are synchronized. We assume that for any global array index i each process r can efficiently (that is, in constant time) compute both: the rank of the target process that owns the corresponding block of A and B, and the local index in the block. This can easily be done for regular distributions of the A and B arrays. Note, however that the correctness of our implementations does not require any specific distribution. Each iteration in Step 1 performs up to four MPI_Get operations, namely to access array elements $A[j-1]$, $B[k]$ and $A[j]$, $B[k-1]$. This can be optimized further by aggregating two or more accesses, if they are located on the same process. For ease of exposition, we do not discuss such (minor) improvements.

3.1 Active Target Synchronization with an Upper Bound of Fences

The first implementation variant uses active target synchronization via a collective *fence* operations. Each iteration of Algorithm 1 becomes a global access epoch, which is surrounded by MPI_Win_fence for each window. In an epoch, each process performs up to four remote memory accesses with MPI_Get. The actual number of iterations that is needed to determine the co-ranks is data-dependent. Therefore, the processes do not necessarily perform the same number of iterations. This is a problem because the collective MPI_Win_fence operation must be called by all processes. One solution is shown in Algorithm 2: it imposes a worst-case upper bound on the number of epochs. Processes that complete the co-ranking procedure early, perform empty epochs to keep in sync with the remaining, potentially still active, processes. An upper bound on the number of iterations is $\lceil \log_2(\min(m, n)) \rceil + 1$.

```
j = min(i,m); k = i-j; j_low = max(0,i-n);
upper = ceil(log2(min(m,n)))+1; active = 1;
do {
   MPI_Win_fence(MODE_NOPUT|MODE_NOPRECEDE);   // (1) start access epoch
   // on both A and B window (not shown)
   if (j>0) a1=GET(A[j-1]); if (k<n) b1=GET(B[k]);
   if (k>0) b2=GET(B[k-1]); if (j<m) a2=GET(A[j]);
   MPI_Win_fence(MODE_NOSTORE|MODE_NOSUCCEED); // (2) end access epoch
   if (j>0&&k<n&&a1>b1) {
      delta = (1+j-j_low)/2;
      k_low = k;
      j -= delta; k += delta;
   } else if (k>0&&j<m&&b2>=a2) {
      delta = (1+k-k_low)/2;
      j_low = j;
      k -= delta; j += delta;
   } else active = 0;
   upper--;
} while (active);
// execute epochs until upper bound is reached
while (upper-- > 0) {
   MPI_Win_fence(MODE_NOPUT|MODE_NOPRECEDE);   // mimic (1)
   MPI_Win_fence(MODE_NOSTORE|MODE_NOSUCCEED); // mimic (2)
}
```

Fig. 2. Co-ranking using collective fences and an upper bound on number of iterations.

The `GET` functionality determines both the target process and the local index on that process for a given global index. It calls MPI_Get to remotely access this element. Since all (local and remote) accesses to the input arrays are read-only, we use the MPI assertions `MPI_MODE_NOPUT` and `MPI_MODE_NOSTORE` as optimization hints to the MPI library. The additional `MPI_MODE_NOPRECEDE` and `MPI_MODE_NOSUCCEED` assertions indicate that there is no active epoch before the opening and after the closing fence. An optimization not considered here would use only a single fence between iterations, but the last assertion pair could be expected to ensure this behavior. With the Blue Gene/P MPI implementation however, there was no performance difference whether these assertions were used or not, which suggests some room for improvement within the MPI library.

3.2 Active Target Synchronization with Global Reduction

Our first implementation variant uses a precalculated upper bound on the number of iterations. However, this is a theoretical worst case and might in practice be too large. If all processes find their co-ranks faster, all extraneous fences consume unnecessary time. To avoid these superfluous fences, we determine at the end of each epoch whether all processes have finished co-ranking. This is accomplished by an MPI_Allreduce at the end of each iteration. Each process contributes its local active flag. Local flag values are combined with a logical "or", and the result tells every process whether there are still active processes. This variant is shown in Algorithm 3. Everything in `[...]` is as in Algorithm 2.

```
[...]
active = 1;
do {
   MPI_Win_fence(MODE_NOPUT|MODE_NOPRECEDE);   // (1) start access epoch
   if (j>0) a1=GET(A[j-1]); if (k<n) b1=GET(B[k]);
   if (k>0) b2=GET(B[k-1]); if (j<m) a2=GET(A[j]);
   MPI_Win_fence(MODE_NOSTORE|MODE_NOSUCCEED); // (2) end access epoch
   [...]
   MPI_Allreduce(MPI_IN_PLACE,&active,1,MPI_INT,MPI_LOR,MPI_COMM_WORLD);
} while (active);
```

Fig. 3. Active one-sided variant with MPI_Allreduce to determine termination.

This variant can never use more iterations than the first implementation. However, whenever the number of iterations of some rank is close to the upper bound, the extra MPI_Allreduce calls are "pure overhead". Unfortunately, without doing the actual co-ranking, there is no way to tell in advance whether Algorithm 2 or 3 is preferable. The MPI standard might be able to help with the general problem exposed by this example, namely to detect epochs where there is no communication activity. A fence operation could report back whether the epoch had any one-sided communication activity – in many cases, the MPI library implementation would have to detect this internally anyway.

Both variants so far have the drawback of adding a collective call to each iteration of the co-ranking algorithm, thus increasing the worst-case complexity by a factor reflecting the time for a collective fence and global reduction operation.

3.3 Passive Target Synchronization with Shared Locks

The third implementation variant uses passive target synchronization. The advantage here is that no collective operations as in the previous variants are required. Since we need the actual data directly after the GET call, each MPI_Get is encapsulated by MPI_Win_lock() and MPI_Win_unlock() operations. All remote accesses are read operations, thus we can allow concurrent accesses by specifying the lock to be shared. Although this implementation involves lock overhead, the processes can now work and terminate independently. The optional assertion MPI_MODE_NOCHECK indicates to the MPI library that accesses are not conflicting (shared and exclusive). On the Jugene system (see Section 4), this assertion improves the performance of the co-ranking by up to 640% at $32k$ MPI processes.

```
double GET(global_pos, window)
{
   target_rank = ...global_pos...; // compute rank from global index
   local_pos = ...global_pos...;   // compute local index
   MPI_Win_lock(MPI_LOCK_SHARED,target_rank,MPI_MODE_NOCHECK,window);
   MPI_Get(result,1,MPI_DOUBLE,target_rank,local_pos,1,MPI_DOUBLE,window);
   MPI_Win_unlock(target_rank,window);
   return result;
}
```

Fig. 4. The GET functionality for the lock variant implementation.

The MPI 2.2 one-sided model allows to perform the lock on only one process at a time, which limits concurrency between MPI_Get calls within each iteration. A different interface might potentially yield better performance. One-sided communication interfaces such as ARMCI [8] or SHMEM [9] define their one-sided communication operations to be explicitly blocking or nonblocking, which gives further opportunities to increase concurrency as discussed in the next section.

3.4 MPI 3.0

The proposed MPI 3.0 standard (available at www.mpi-forum.org) considerably extends the MPI 2.2 one-sided communication model and does address some of the problems discussed above. In particular, it introduces new MPI_Rget and MPI_Rput one-sided communication operations, which return a request object. Inside the epoch it is possible to complete such operations by issuing an MPI_Wait on this request. With this feature, the whole merge algorithm could be performed in a single MPI_Win_fence epoch. Each process would independently iterate, and enforce completion of the MPI_Rget calls in each iteration.

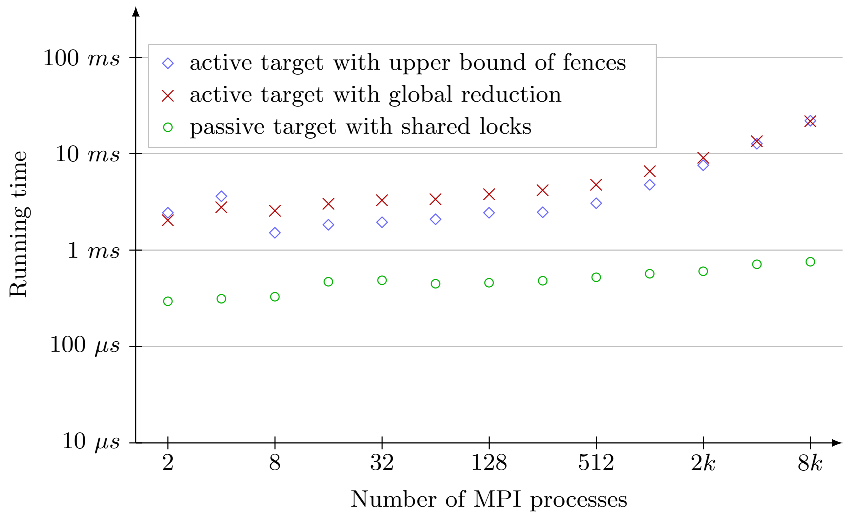

Fig. 5. Co-ranking performance with the three implementation variants, weak scaling.

4 Experimental Results

We have implemented the distributed, stable merging algorithm with all three alternatives for the co-ranking step. The implementations have been evaluated on a large distributed-memory supercomputer: the "Jugene" IBM Blue Gene/P installation in Jülich/Germany with 73,728 nodes, each equipped with a 4-way PowerPC processor (850 MHz) and 2 GiB memory.

All experiments used double-precision floating-point elements with sorted random inputs, and were conducted in SMP mode with one MPI process per node. We performed both strong and weak scaling experiments. Figure 5 shows weak scaling results for the three co-ranking implementation variants. The two input arrays have 10 and 20 million elements per process, respectively. Therefore, the total number of elements is p times these local sizes. Since the number of iterations of the co-ranking algorithm is logarithmic in the minimum number of elements, we would expect its running time for an increasing number of processes to grow with at least $\mathcal{O}(\log p)$. The passive target variant seems to achieve this slow growth while the curves for the two active target variants show a steeper ascent. Both variants with the collective fence synchronization are by far slower than the lock variant, reaching a factor of 29 difference at 8,192 processes. We therefore choose the lock variant as our implementation for the co-ranking step.

In Figures 6 and 7, we present the individual times for the three main steps of the complete merge algorithm: co-ranking, copying of remote data, and local merge. We determined the running time of the local merge $T_{\text{seqmerge}}(n, m)$ to be $0.0484 \cdot (n + m)$ μsec. We use this sequential time to calculate the parallel efficiency of our merge implementation $E(n, m, p)$ as $T_{\text{seqmerge}}(n, m)/(p \cdot T_{\text{parmerge}})$, which is given as a percentage above the total running time.

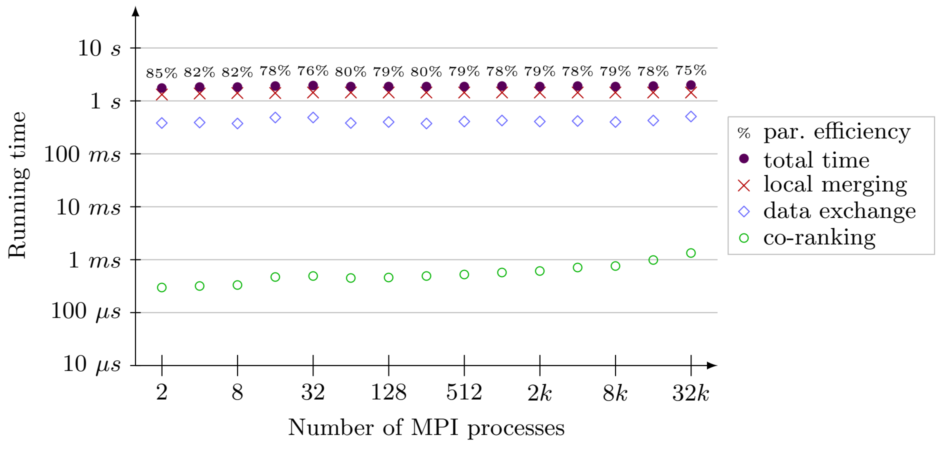

Fig. 6. Merging performance on Jugene, weak scaling.

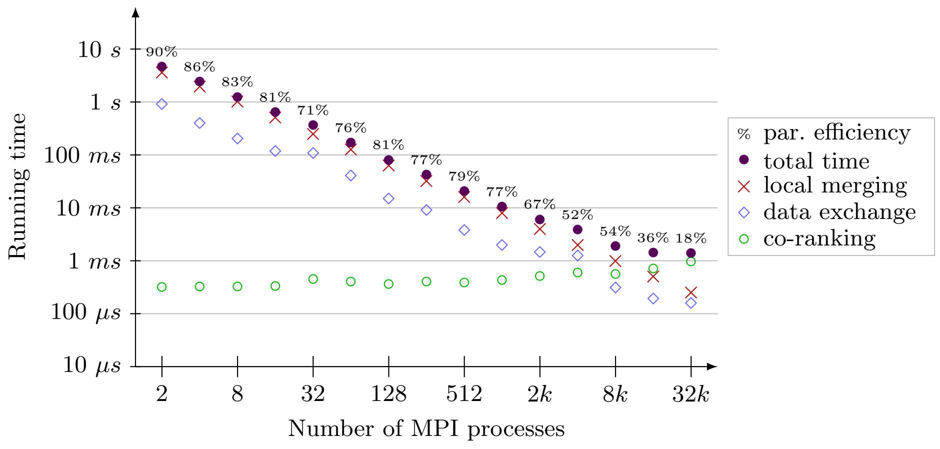

Fig. 7. Merging performance on Jugene, strong scaling.

Figure 6 shows the weak-scaling behavior for the individual steps of the parallel merge algorithm, including its total running time. Note that the co-ranking step is indeed about a thousand times faster than the data copy and local merge. With a fixed input size per process, we would expect the running time for the data copy and local merge to remain constant, which is indeed the case. Only the time of the co-rank step increases very slowly with p. The overall parallel efficiency stays at around 80%.

Figure 7 presents results from a strong scaling experiment. The total number of elements for the two input arrays are $32 \cdot 2^{20}$ and $48 \cdot 2^{20}$, respectively. This means that we always use only 640 MiB of input data and distribute this over an increasing number of processes. Even with such a relatively small amount of data (note that this Blue Gene/P system has only 2 GiB of memory available per node), our lock-based co-ranking implementation scales up to 32,768 processes, where only a few thousand input elements exists per process, albeit with decreasing efficiency from around $2k$ processes.

5 Summary and Outlook

We presented a stable, distributed-memory parallel merge algorithm, and in particular discussed implementation alternatives in the MPI 2.2 and MPI 3.0 one-sided communication models. The alternatives have been implemented and we reported on initial experiments on a Blue Gene/P system. To our surprise, the lock-based variant used for the co-ranking preprocessing step showed considerably better performance than the other possibilities considered. However, this still needs stronger experimental support, and we are continuing the experimental work with the distributed merge algorithm.

Acknowledgments. The authors want to thank the anonymous reviewers for the valuable comments, Harald Klimach for his input on the manuscript, and Marc-André Hermanns for his ideas concerning the one-sided implementations.

References

1. S. G. Akl and N. Santoro. Optimal parallel merging and sorting without memory conflicts. *IEEE Transactions on Computers*, C-36(11):1367–1369, 1987.
2. N. Deo, A. Jain, and M. Medidi. An optimal parallel algorithm for merging using multiselection. *Information Processing Letters*, 50(2):81–87, 1994.
3. N. Deo and D. Sarkar. Parallel algorithms for merging and sorting. *Information Sciences*, 56(1–3):151–161, 1991.
4. A. V. Gerbessiotis and C. J. Siniolakis. Merging on the BSP model. *Parallel Computing*, 27(6):809–822, 2001.
5. T. Hagerup and C. Rüb. Optimal merging and sorting on the EREW PRAM. *Information Processing Letters*, 33:181–185, 1989.
6. J. Katajainen, C. Levcopoulos, and O. Petersson. Space-efficient parallel merging. *Informatique Théoretique et Applications*, 27(4):295–310, 1993.
7. MPI Forum. *MPI: A Message-Passing Interface Standard. Version 2.2*, September 4th 2009. www.mpi-forum.org.
8. J. Nieplocha, V. Tipparaju, M. Krishnan, and D. K. Panda. High performance remote memory access communication: The ARMCI approach. *International Journal on High Performance Computing Applications*, 20(2):233–253, 2006.
9. S. W. Poole, O. Hernandez, J. A. Kuehn, G. M. Shipman, A. Curtis, and K. Feind. OpenSHMEM - toward a unified RMA model. In D. A. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1379–1391. Springer-Verlag, 2011.
10. Y. Shiloach and U. Vishkin. Finding the maximum, merging and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.
11. P. J. Varman, B. R. Iyer, D. J. Haderle, and S. M. Dunn. Parallel merging: algorithm and implementation results. *Parallel Computing*, 15(1-3):165–177, 1990.