Parallel Prefix (Scan) Algorithms for MPI

Peter Sanders[1] and Jesper Larsson Träff[2]

[1] Universität Karlsruhe
Am Fasanengarten 5, D-76131 Karlsruhe, Germany
sanders@ira.uka.de
[2] C&C Research Laboratories, NEC Europe Ltd.
Rathausallee 10, D-53757 Sankt Augustin, Germany
traff@ccrl-nece.de

Abstract. We describe and experimentally compare three theoretically well-known algorithms for the parallel prefix (or *scan*, in MPI terms) operation, and give a presumably novel, doubly-pipelined implementation of the in-order binary tree parallel prefix algorithm. Bidirectional interconnects can benefit from this implementation. We present results from a 32 node AMD Cluster with Myrinet 2000 and a 72-node SX-8 parallel vector system. On both systems, we observe improvements by more than a factor two over the straight-forward binomial-tree algorithm found in many MPI implementations. We also discuss adapting the algorithms to clusters of SMP nodes.

Keywords: Cluster of SMPs, collective communication, MPI implementation, prefix sum, pipelining.

1 Introduction

The *parallel prefix* or *scan* operation is a surprisingly versatile primitive and a basic building block in massively parallel algorithms for a variety of different problems, as shown by research in the 80ties and 90ties [2, 5]. Scan primitives are also included among the collective operations of the *Message Passing Interface* (MPI) [10], as an *inclusive* operation `MPI_Scan`, and with the MPI-2 standard [3], also as an *exclusive* operation `MPI_Exscan`.

The parallel prefix operation can be explained as follows. Let p be the number of processing element(PE)s numbered consecutively from 0 to $p-1$, and let a sequence of p elements x_i with an associative, binary operation $\oplus$ be given.

The *inclusive parallel prefix operation* computes for each PE j, $0 \leq j < p$ the value $\bigoplus_{i=0}^{j} x_i = x_0 \oplus x_1 \oplus \cdots \oplus x_j$, with the convention that $\bigoplus_{i=j}^{j} x_i = x_j$ (a one element sum is just that one element).

The *exclusive parallel prefix operation* computes for each PE j, $0 < j < p$ *except* PE 0 the value $\bigoplus_{i=0}^{j-1} x_i$. With this definition, no neutral element for the operation $\oplus$ is required.

For use with the scan (and other reduction) collectives, MPI provides a number of standard, binary operations like summation, maximum, boolean and bitwise operations etc. on standard datatypes like integers, doubles, and so forth. In

addition the user can define new associative operations on arbitrarily structured, possibly even non-contiguous datatypes. Instead of a parallel prefix on a single element per process, the MPI scan operations work element-wise on vectors of elements. The number of elements in the vector is given by a `count` argument in the call of `MPI_Scan`/`MPI_Exscan`.

2 The scan algorithms

In this section we describe three standard algorithms for the parallel prefix operations, and discuss their implementation in MPI. We focus exclusively on the inclusive scan operation, but the discussion applies *mutatis mutandis* to `MPI_Exscan`. We assume single-ported communication in a fully connected network. Communication cost is $\alpha + \beta m$ for a communication involving m data elements. We use three variants of this model: a) *half-duplex* where each communicating PE can either send or receive a message, b) *telephone model*, where a matched pair of PEs can communicate bidirectionally, and c) *full-duplex* where a PE can simultaneously send data to one PE and receive data from a possibly different PE. An m element $\oplus$ computation takes γm time of local work.

All algorithms will work in (implicitly) synchronized rounds and exchange data packets of equal length. Hence, most of the time it suffices to discuss the number of communication rounds and the amount of data sent and received by each PE. We also discuss the total communication volume.

We use the shorthand $\oplus[j..k]$ for $\bigoplus_{i=j}^{k} x_i$.

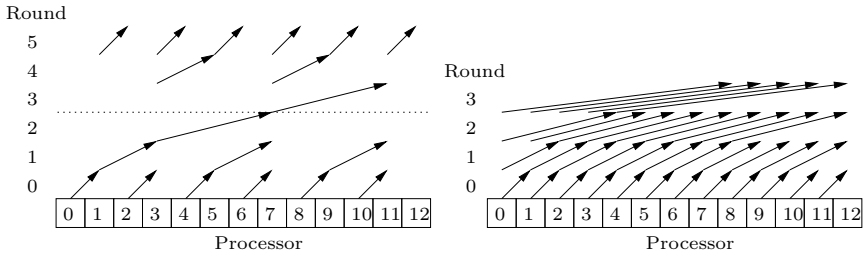

Fig. 1. The communication patterns for the simple binomial (left) and the simultaneous binomial (right) tree algorithm for $p = 13$.

2.1 Binomial tree

Let $n = \lfloor \log_2 p \rfloor$. The binomial tree algorithm consists of an *up-phase* and a *down-phase* each of n rounds. In round k, $k = 0, \ldots, n-1$ of the up-phase each PE j satisfying $j \wedge (2^{k+1} - 1) = 2^{k+1} - 1$ (where $\wedge$ denotes "bitwise and") receives a partial result from PE $j - 2^k$ (provided $0 \leq j - 2^k$). Afterwards,

PE $j - 2^k$ is inactive for the remainder of the up-phase. The receiving PEs add the partial results, and after round k have a partial result of the form $\oplus[j - 2^{k+1} + 1..j]$. In the down-phase we count rounds downward from n to 1. A PE j with $j \wedge (2^k - 1) = 2^k - 1$ sends its partial result to PE $j + 2^{k-1}$ (provided $j + 2^{k-1} < p$) which can now compute its final result $\oplus[0..j + 2^{k-1}]$. The communication pattern is shown in Figure 1.

The number of communication rounds is $2\lfloor \log p \rfloor$, and the total communication volume is bounded by $2pm$ since in each round half the PEs become inactive. Since each PE is either sending or receiving data in each round, with no possibility for overlapping of sending and receiving due to the computation of partial results, the algorithm can be implemented in the half-duplex model.

2.2 Simultaneous binomial tree

Starting from round $k = 0$, in round k, PE j sends its partial result to PE $j + 2^k$ (provided $j + 2^k < p$) and receives a partial result from PE $j - 2^k$ (provided $0 \geq j - 2^k$). The partial results are added. It is easy to see that after round k, PE js partial result is $\oplus[\max(0, j - 2^{k+1} + 1)..j]$. PE j can terminate when both $j - 2^k < 0$ (nothing to receive) and $j + 2^k \geq p$ (nothing to send). This happens after $\lceil \log p \rceil$ rounds. This algorithm goes back (at least) to [4], and is illustrated in Figure 1.

The total communication volume is bounded by $\lceil \log p \rceil pm$ since (almost) all PEs are active in all rounds. Since each PE is both sending and receiving data from two different PEs, the analysis assumes the full-duplex model. In [6] it is shown that the algorithm can be generalized to exploit k-ported communication.

A different algorithm with the same characteristics, but based on a butterfly communication pattern is used in the `mpich2` MPI implementation. For this algorithm the telephone model of communication suffices but it has unbalanced computational load — in each round, half the PEs compute two partial results.

2.3 Pipelined binary tree

The third algorithm arranges the PEs in a binary tree T with in-order numbering. This numbering has the property that the PEs in the subtree $T(j)$ rooted at j have consecutive numbers in the interval $[\ell, \ldots, j, \ldots, r]$ where ℓ and r denote the first and last PE in the subtree $T(j)$, respectively. The algorithm has two phases. In the *up-phase*, PE j first receives the partial result $\oplus[\ell..j-1]$ from its left child and adds x_j to get $\oplus[\ell..j]$. This value is stored for the down-phase. PE j then receives the partial result $\oplus[j+1..r]$ from its right child and computes the partial result $\oplus[\ell..r]$. PE j sends this value upward without keeping it. In the *down-phase*, PE j receives the partial result $\oplus[0..\ell-1]$ from its parent. This is first sent down to the left child and then added to the stored partial result $\oplus[\ell..j]$ to form the final result $\oplus[0..j]$ for j. This final result is sent down to the right child.

With the obvious modifications, the general description covers also nodes that need not participate in all of these communications: Leaves have no children.

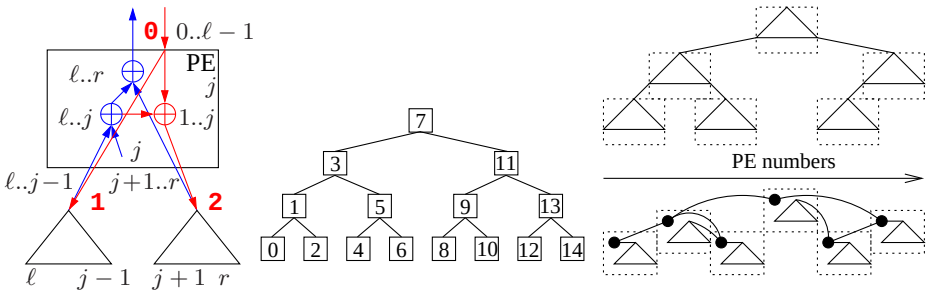

Fig. 2. From left to right: The basic schedule of the doubly pipelined algorithm. A balanced binary tree with in-order numbering. Two ways to build a binary tree from a cluster of six SMPs.

Some nodes may only have a leftmost child. Nodes on the path between root and leftmost leaf do not receive data from their parent in the down-phase. Nodes on the path between rightmost child and root do not send data to their parent in the up-phase. The communication pattern and examples of trees are shown in Figure 2.

Let the *height* n of the tree denote the length of the longest root-to-leaf path. The number of rounds for both up- and down-phases are at most $2n-1$ each. The total communication volume per phase is bounded by $(p-1)m$. The algorithm assumes only half-duplex communication.

It is a standard observation (e.g. [7]) that each PE is (in each phase) only active in three consecutive rounds. Hence, successive up-phases (and down-phases) can be *pipelined*. More specifically, if the m element vectors can be divided into b blocks (and the operation $\oplus$ on the m element vectors can likewise be blocked), each phase can be done in $3(b-1)+2n-1$ rounds: the $2n-1$ rounds is the delay for the first block delivered at the root (or at the lowest leaf), with a new block delivered at every third round. Since the partial results computed by PE j in the up-phase is either needed by j or immediately sent upwards, there is no need for intermediate buffering between up- and down-phases. A single buffer of size $O(m/b)$ for receiving a single intermediary block therefore suffices also for the pipelined implementation. In our cost model, an m element prefix sum can be computed in time $O(n+m)$ using an optimal block size of $\Theta(\sqrt{m/n})$. Note that using a balanced binary tree we have $n = \lceil \log(p+1) \rceil - 1$.

For bidirectional communication networks in the telephone model the two pipelined phases can be combined. This can reduce the number of rounds by up to a factor of two. Depending on its position in the tree, a PE will first perform a certain number d of rounds working only on upward traffic while waiting for the first packet of downward data.[3] After this *fill phase*, it enters into a *steady state* such that in each round it exchanges one block of data with its parent

[3] In a complete binary tree, this waiting time is proportional to the number of parent connections one has to follow until reaching the leftmost root-leaf path in the tree.

or one of its children. After $3b - d$ rounds in steady state, the up-phase blocks have been completed, and in d rounds of the *drain phase* the last blocks of the down-phase are processed. We call this algorithm the *doubly pipelined prefix algorithm*.

The largest *delay* is incurred for the rightmost leaf PE in the binary tree, which has to wait for $2(2n-1)$ rounds for the first block to arrive. A new block arrives every third round, so the total number of rounds becomes $3(b-1)+4n-2$, or almost a factor two better than the $6(b-1)+4n-2$ rounds required for the up- and down-phase of the two-phase algorithm.

Instead of pipelining, the same asymptotic running time is achieved by the algorithm in [1] which by repeated halving splits the m elements into p blocks, on which simultaneous scans are carried out by edge-disjoint binomial trees. This algorithm assumes that p is a power of two (with a trivial generalization), and also in terms of constant factors the algorithm is worse than the doubly pipelined prefix algorithm.

3 Performance evaluation

The algorithms from Section 2 have currently been implemented for the case of one MPI process per node. The algorithms have been benchmarked on both a 32-node AMD cluster with Myrinet 2000, and the 72 node SX-8 parallel vector supercomputer at HLRS (Hochleistungsrechenzentrum Stuttgart, Germany). We compare four algorithms , namely

- binomial tree
- simultaneous binomial trees
- pipelined binary tree
- doubly pipelined binary tree

Results are shown in Figure 3 which shows the achieved throughput as a function of problem size m for fixed number of processes. For the pipelined algorithms, the block size has been chosen proportional to $\sqrt{m/n}$ with experimentally determined constants depending on α and β.

On both systems and all input sizes, all algorithms are dominated by just two algorithms. The simultaneous binomial tree algorithm is best for small message lengths (m up to about 10KBytes for the Myrinet cluster, and up to about 1MByte for the SX-8 system). For very small messages, it is up to a factor of two better than all other algorithms and it dominates the plain binomial tree algorithm for all input sizes. Beyond this threshold, both pipelined algorithms give better throughput, although the difference between the pipelined and the doubly pipelined algorithms is smaller than expected from the theoretical analysis. Nevertheless, the capability for full-duplex communication can be exploited. This is especially clear for the SX-8 system.

On the Myrinet cluster, all algorithms suffer a performance degradation for message sizes beyond 2MB. This effect requires further investigation.

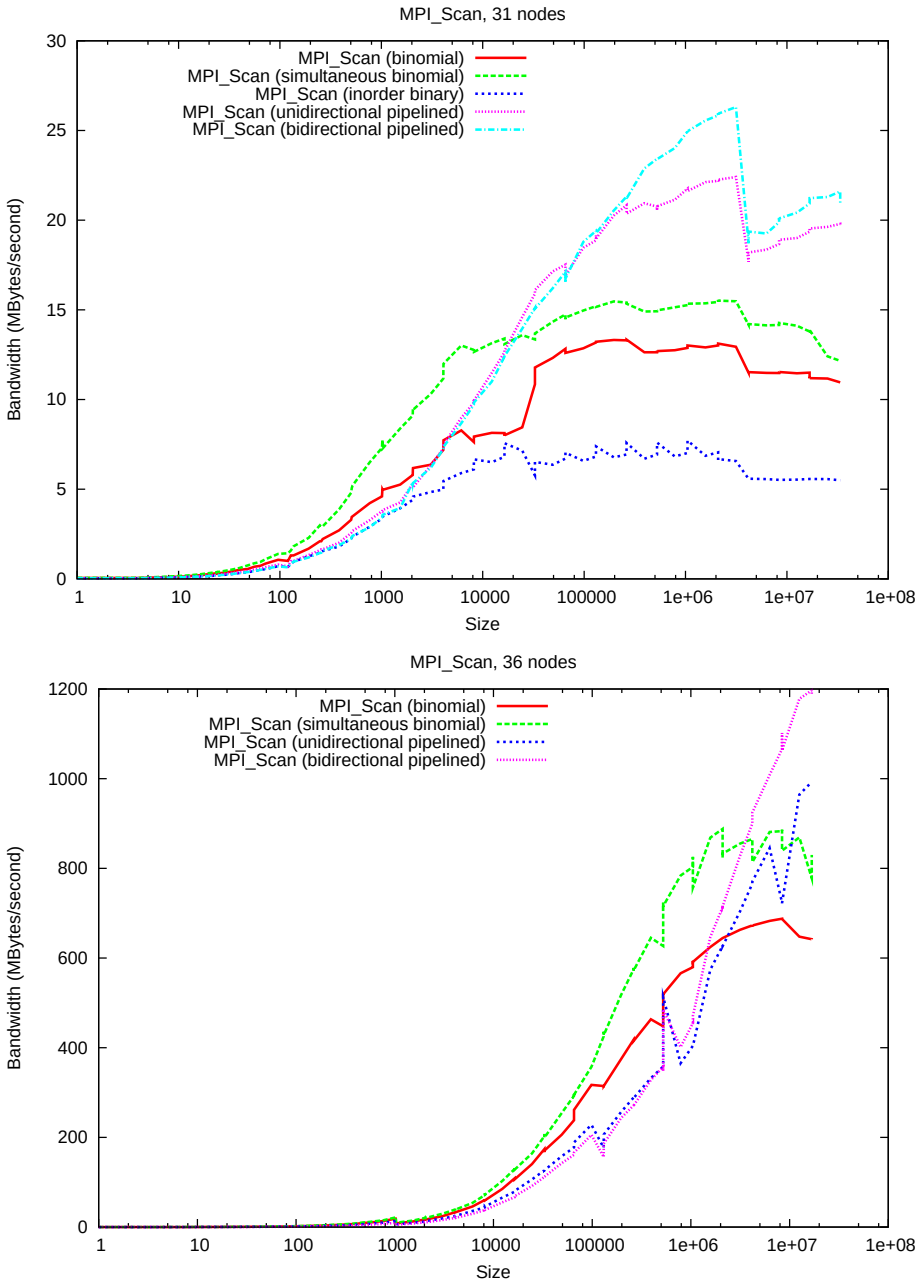

Fig. 3. The four scan algorithms binomial trees, simultaneous binomial trees, pipelined binary tree, doubly pipelined binary tree. Top: 31-node Myrinet cluster (with add. measurements for nonpipelined binary tree); Bottom: 36 nodes of the NEC SX-8.

4 Adaptation to the SMP case

The parallel prefix algorithms were developed assuming a homogeneous communication network. For clusters of SMP nodes this assumption does not hold, and severe node contention can result if many PEs per SMP node must in the same round send and/or receive data from other nodes. In particular the simultaneous binomial tree algorithm will inevitably suffer from this kind of node contention.

Now we discuss several possible improvements for the case that there are P *SMP* (nodes) with p_i consecutively ranked PEs in SMP i.

For small inputs and/or very slow inter-SMP communication, a simple hierarchical decomposition works well: First compute a parallel prefix within each SMP. Then perform a parallel (exclusive) prefix over the SMPs using the result of the last PE on each SMP. Finally, within each SMP, add the global result to each local result. This algorithm has the advantage that at any time at most one PE per SMP is performing inter-SMP communication.

For large inputs it is better to arrange all the PEs into a single tree taking care that inter-SMP communication is small. This way, the time for intra-SMP prefix computation will not appear in the term of the execution time that depends on the input size m. We propose two basic ways to do this which are depicted in the right part of Figure 2: One is to build local trees of depth $O(\log p_i)$ on each SMP and to build a binary tree of local trees as follows: The root PE of a left successor in the SMP tree has the leftmost PE of its parent SMP as its parent in the PE tree. Analogously, a right successor has a rightmost PE as its parent. Now suppose $p_i = p/P$ for all SMPs. We get a PE tree of height $(1+o(1)) \log \frac{n}{p} \log P$. At most three PEs in each SMP perform inter-SMP communication. The total volume of inter-SMP communication is $\leq 2mP$.

At the cost of increasing the inter-SMP communication to about $3mP$, we can decrease the height of the tree to $\lceil \log(P+1) \rceil + \max_{0 \leq i < P} \lceil \log p_i \rceil - 1$ and reduce the number of PEs with inter-SMP communication to at most two per SMP: The leftmost PEs g_i of each SMP form a *global* balanced binary tree of height $h = \lceil \log(P+1) \rceil - 1$ with the following properties: An in-order traversal meets growing PE numbers. Only leaves have no left successor. All PEs without a right child are only on the rightmost path through the tree. The remaining PEs of each SMP form a *local* tree of height $\lceil \log p_i \rceil - 1$ rooted at some node r_i. If global tree PE g_i has no right successor in the global tree, its right successor is r_i. If g_i for $i > 0$ has no left successor in the global tree, its left successor will be r_{i-1}. It is easy to verify that the resulting tree has the claimed properties.

5 Summary

We described and implemented three algorithm for the MPI scan collective. As shown by the performance evaluation, a production quality MPI should use a hybrid approach, using the simultaneous binomial tree algorithm for small problems, while switching to the doubly pipelined algorithm for large scan problems. To the best of our knowledge our implementation is the first implementation of

a pipelined scan algorithm. The doubly pipelined algorithm is new. Efficiently mapping communication trees to SMPs is already described for broadcasting in [8]. However our method to maintain the canonical numbering of the PEs as an in-order numbering of the tree is new.

For the design and determination of block sizes a linear cost function was assumed. This is a simplified assumption, and potentially more accurate cost models exist. In [9] the prefix-sums problem is studied in the LogP model from a different perspective (what is the largest number of x_is that can be reduced in a given time?). The resulting algorithms are complex, so there is a trade-off between accuracy and implementation concerns.

References

1. S. Bae, D. Kim, and S. Ranka. Vector prefix and reduction computation on coarse-grained, distributed memory machines. In *International Parallel Processing Symposium/Symposium on Parallel and Distributed Processing (IPPS/SPDP 1998)*, pages 321–325, 1998.
2. G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.
3. W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI – The Complete Reference*, volume 2, The MPI Extensions. MIT Press, 1998.
4. W. D. Hillis and J. Guy L. Steele. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.
5. J. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.
6. Y.-C. Lin and C.-S. Yeh. Efficient parallel prefix algorithms on multiport message-passing systems. *Information Processing Letters*, 71:91–95, 1999.
7. E. W. Mayr and C. G. Plaxton. Pipelined parallel prefix computations, and sorting on a pipelined hypercube. *Journal of Parallel and Distributed Computing*, 17:374–380, 1993.
8. P. Sanders and J. F. Sibeyn. A bandwidth latency tradeoff for broadcast and reduction. *Information Processing Letters*, 86(1):33–38, 2003.
9. E. E. Santos. Optimal and efficient algorithms for summing and prefix summing on parallel machines. *Journal of Parallel and Distributed Computing*, 62(4):517–543, 2002.
10. M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI – The Complete Reference*, volume 1, The MPI Core. MIT Press, second edition, 1998.