High Performance Computing

Reduction trees for MPI Reductions

Project 2

Johannes Winklehner 1226104

Armin Friedl 1053597

June 24, 2016

Contents

1 Problem Description **2**

2 Implemented Algorithms **2**

2.1 Binomial Tree Reduce . . . 3

2.2 Fibonacci Tree Reduce . . . 4

2.3 Binary Tree Reduce . . . 5

3 Results **5**

3.1 Process Count . . . 6

3.2 Array Size . . . 6

4 Analysis **9**

5 Appendix **10**

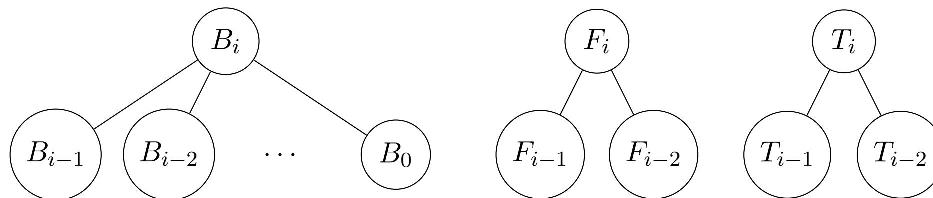
1 Problem Description

The purpose of this project is to compare different implementations of the collective communication call MPI_Reduce. The compared implementations should all use different forms of Tree Reduction algorithms. As a baseline for the comparison serves a given implementation of the MPI standard, which is in our case NEC MPI.

Binomial Tree A binomial tree has a non-fixed degree where each tree B_i has exactly i subtrees of size B_0 to B_{i-1}. The number of nodes in such a tree is equal to 2^i and the depth is i.

Fibonacci Tree The Fibonacci tree uses a fixed degree of 2 where a tree of size F_i has one subtree of size F_{i-1} and one of F_{i-2}. Therefore the number of nodes in this kind of tree is $fib(i+3)-1$ using the Fibonacci function $fib(x) = fib(x-1) + fib(x-2)$ and its depth is as well i.

Binary Tree The binary tree used for reduction is a common complete binary tree where a tree T_i has two subtrees T_{i-1}. Such a tree has $2^{i+1}-1$ nodes and its depth is as for the other types i.



All three implementations of the reduce function must use exactly the same interface as the MPI standard defines it. This interface is shown in Listing 1. This requires that all implementations support any arbitrary MPI datatype as well as operations. The standard also provides some constraints regarding the associativity and commutativity of executable operations. Every MPI operation must be associative, but does not necessarily have to be commutative. This means that all results of the operation must be computed in the MPI rank order of all processes.

```
1 int MPI_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
```

Listing 1: MPI Reduce interface

The standard also defines additional features of the reduce function, for example an in place operator for the root process. However since those details where not mentioned in the assignment description, we did not consider them as part of the project.

The basic algorithm for a tree reduction, which will be shown in the next section, is very similar for all kinds of trees and uses Point-to-Point communication between tree nodes. The assumption for our implementations to be efficient is that the underlying communication network is fully connected and allows for bidirectional communication.

2 Implemented Algorithms

The basic algorithm for a tree reduction is very simple and is shown in Algorithm 1. At first the parent and all child nodes have to be determined to know the communication partners of each

process. Then each process receives the partial results from all of its children and calculates its own result from the received data. To ensure the correctness of the result for non commutative operations the iteration of child nodes has to be done in rank order. Processes which are leaf nodes in the tree have no children and therefore skip the receiving part of the algorithm. If a process has a parent and is therefore not the root process, it sends its result to the determined parent node. However if the process is the root process the reduction is finished and can be returned.

Algorithm 1: Tree Reduce

Input: An array $\vec{a}$ of a given *datatype* with size *count* for each process
Output: The result of the reduction on the *root* process

```
1  determine parent and children;
2  result = a⃗;
3  forall child in children do
4  |   receive result from child;
5  |   result = local reduce of received array and result;
6  end
7  if parent exists then
8  |   send result to parent;
9  else
10 |   output = result;
11 end
```

The calculation of the parent and child nodes is the only aspect which has to be changed for all possible kinds of trees. However there are of course certain optimizations possible where some knowledge about the structure of the tree can be used. Such implementation details will be shown in the following part. The code for all our implementations can be found in the Appendix in Section 5.

2.1 Binomial Tree Reduce

The first of the three implementations we completed was the binomial tree reduction. Since there were already some examples and explanations on how reductions and broadcasts work on binomial trees presented during the lectures, this was probably the most straight forward part of the project. When looking at some trees of different sizes we quickly noticed, that the position of each node is static and the tree only grows in one direction. This fact can be used in a sense that the children do not have to be precomputed but instead can be calculated during the loop before the corresponding receive operation. A comparison between a B_2 and a B_3 tree is shown in Figure 1.

From some of those trees we then determined that for the node with rank r the child in each iteration is $r + i$ where $i = 1$ at the start and is multiplied by 2 after each iteration. Before each iteration there is an additional condition, which checks if the node has any children left or if it should send the result.

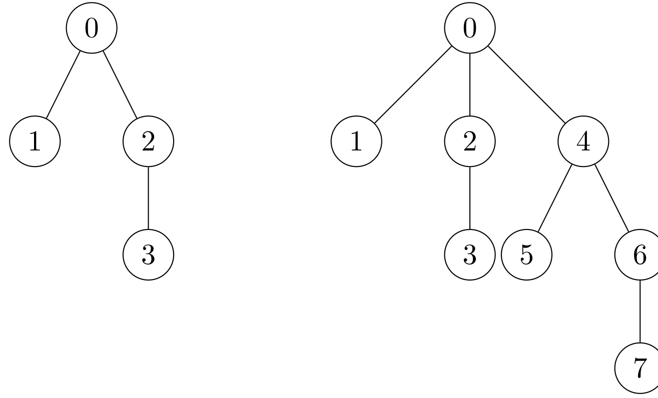


Figure 1: Comparison between a B_2 and a B_3

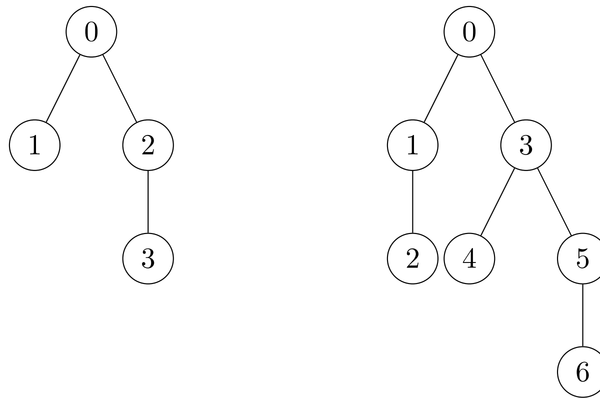


Figure 2: Comparison between a F_2 and a F_3

2.2 Fibonacci Tree Reduce

The core difference of a Fibonacci tree compared to a binomial tree is the fixed degree of 2. To guarantee the correct order of the computed operation the position of a node inside the tree is not only dependent on the rank of the process, but also on the total size of the tree. This is due to the fact that all ranks in one subtree must be lower than the ranks in the second subtree. Therefore the position of a node with a certain rank changes depending on the tree size. This can be seen in the comparison of the trees F_2 and F_3 in Figure 2.

During the receiving step of the algorithm we do not need a loop any more, since there are always two or less children for each node. On the other hand the calculation of the children now has to be done in a loop. As a first step we have to determine the size of the tree which can contain all processes. This can be done by searching the Fibonacci numbers for the first value which is greater than the number of processes. Since we know the size of both subtrees using the Fibonacci numbers we can determine whether a node is supposed to be in the left or right subtree. When doing this recursively the position of a node and its children can be calculated. The runtime of this part depends on the size of the tree and is therefore bound by the Fibonacci numbers.

Now that all communication partners have been determined each process has to execute

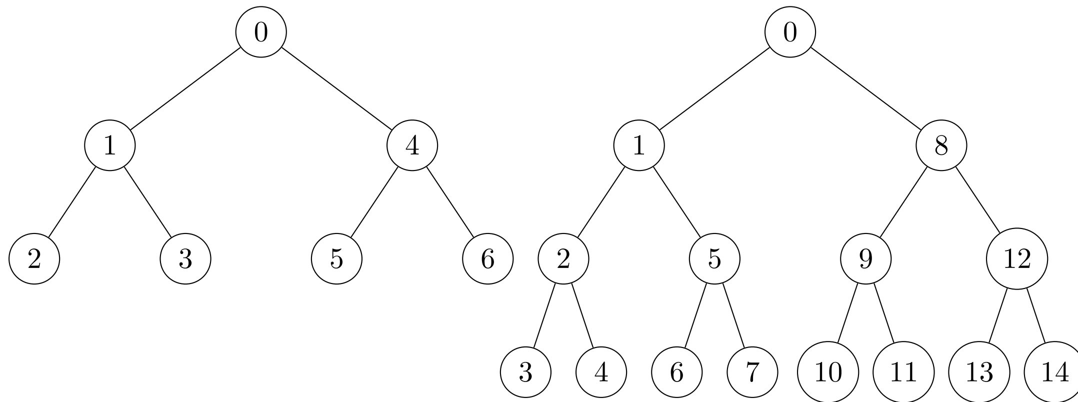


Figure 3: Comparison between a T_2 and a T_3

at most two receives and afterwards one send command. Noticeable when comparing this technique to the biomial tree is that there is already one less node in a tree F_3 than in the B_3. This means that the binomial tree can handle more processes in the same number of rounds.

2.3 Binary Tree Reduce

The reduction using a binary tree can be implemented in a very similar way like the Fibonacci tree since the degree is also two. The key difference is of course the structure of the trees and therefore the calculation of the children. Again the position of certain nodes changes depending on the size of the tree since the lower ranks must be in the left subtree and the higher ones in the right subtree. The structure of such trees can be shown rather nice because they are simply complete binary trees. There is again a comparison between a tree T_2 and T_3 which is shown in Figure 3.

The tree size as well as the computation of the child nodes can be done using a logarithmic function on the number of processes. The rest works in exactly the same way as the previously explained algorithms. When structuring the tree like this the drawback is that in each round a node receives data from both children. As a result the number of rounds for this algorithm is the size of the tree plus an additional round.

3 Results

Before we compared the runtime of our algorithms the correctness has to be tested by a reasonable amount. As already stated in the project description this process can be done easily by comparing each result to the MPI_Reduce function. This can be done for all implementations at once to quickly test them for correctness. With this method we tried various combinations of array sizes, process numbers as well as different operations and the result always turned out to be correct.

After doing those tests to show the correctness we started doing some benchmarking comparisons of our implementations. To do this we utilized 36 nodes of the jupiter system with 16 processes each. We did two kinds of tests which will be explained now to compare the runtime of our implementations as well as the MPI_Reduce function.

3.1 Process Count

For the first benchmark we used a different number of processes to check the scaling of all methods. The size of the array on each process for this test is 1000. This is a rather low value, but since we are using tree reduction which is not optimal for high amounts of data such a value makes sense. The amount of processes used was increased from starting with only one node up to using all available 36 nodes. On each node all 16 processes where used in all tests. Therefore the total process count ranged from 16 to 576. For all out tests in this project we used a repetition count of 30 which allowed us to run a high number of different inputs in a reasonable amount of time. In Figure 5 the results of this benchmark are shown.

3.2 Array Size

Our second used a fixed number of processes but the size of the local arrays was increasing. This should show how the different implementations perform for small arrays or even a single number. But it also shows how they perform with a large amount of data on each process. The amount of nodes was fixed at 36 for the complete test. The size of the local arrays was increased by a factor of 10 in each iteration starting with just 1 and increased it up to 1000000. The number of repetitions is the same as for the last test at 30.

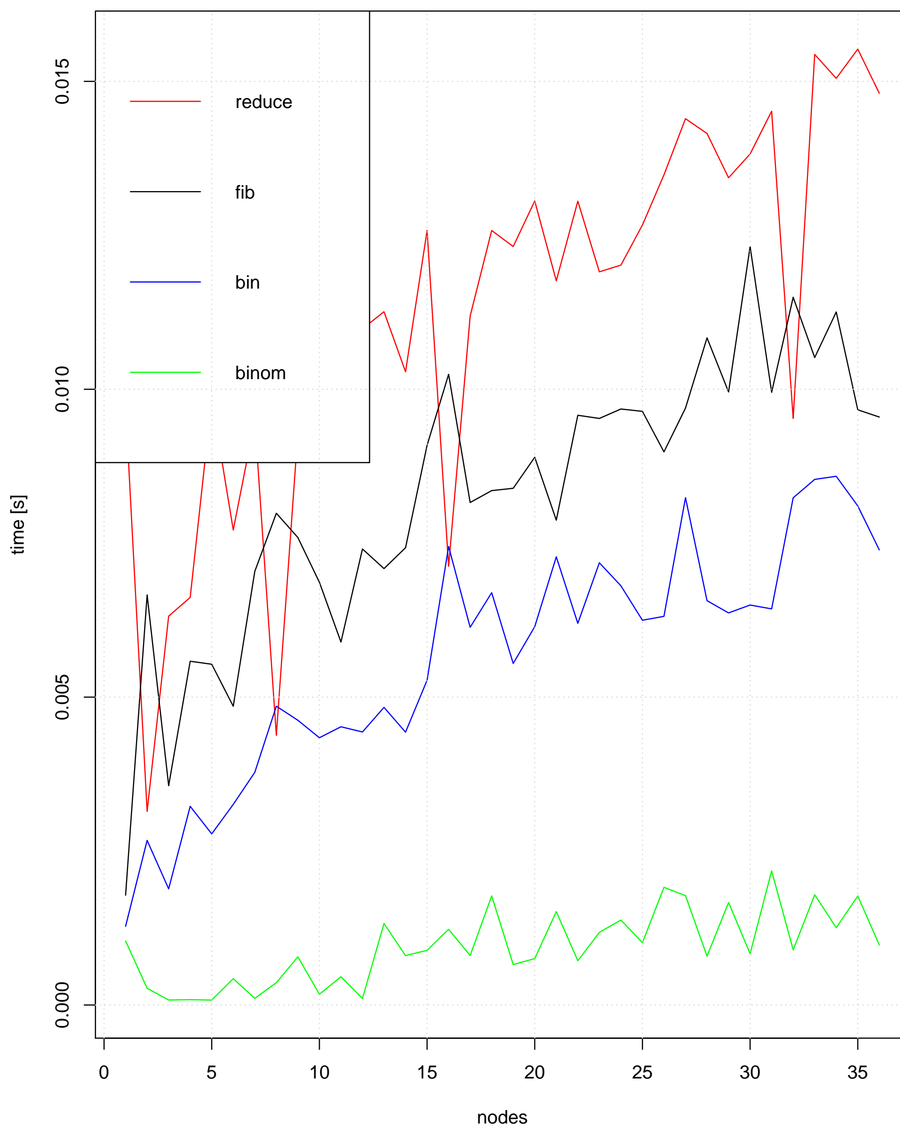


Figure 4: Average runtimes on 1 to 36 nodes with 16 processes each.

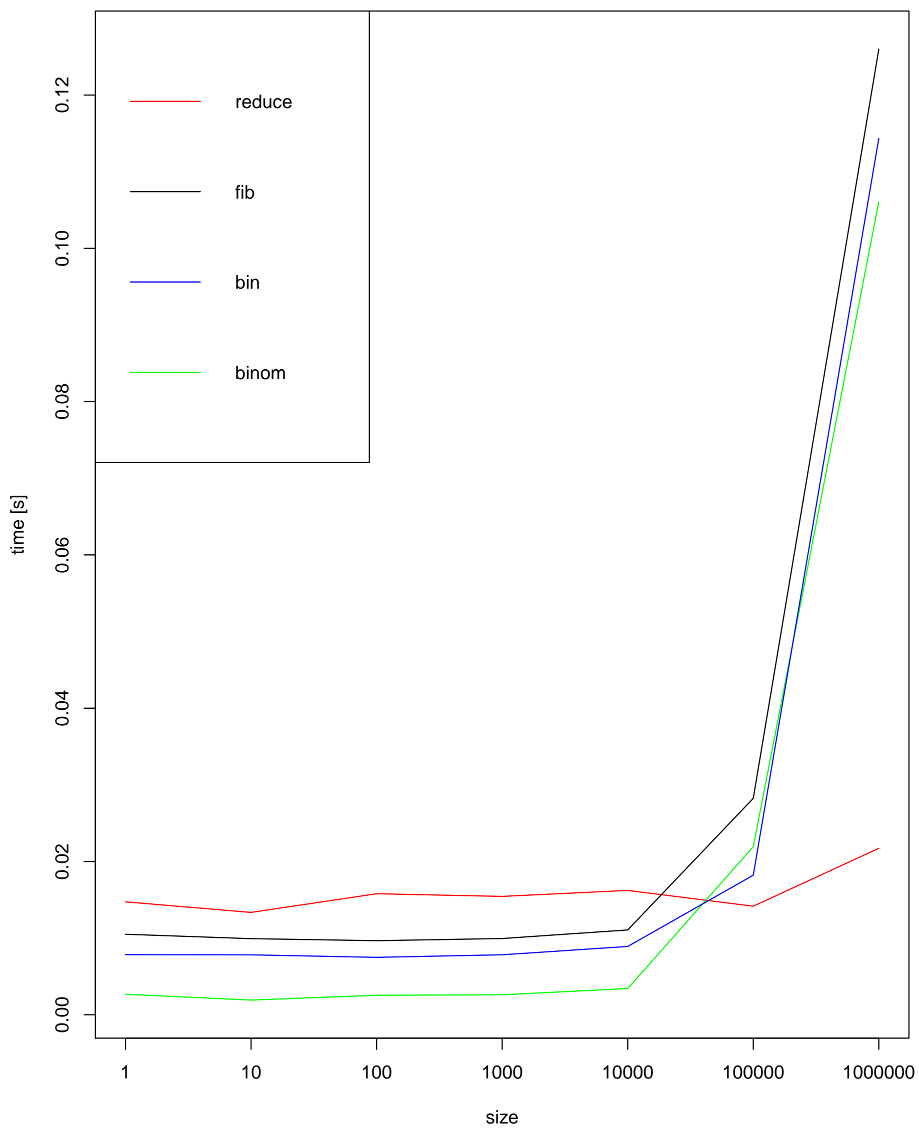


Figure 5: Average runtimes on 36 nodes with an array size of 1 to 1000000.

4 Analysis

Our first result seen in Figure 5 suggests that our implementation using the binomial tree got the best results by a big margin. This result was very surprising to us because we expected that the MPI_Reduce function of the library would outperform our rather simple implementations. However this seems not to be the case and such tree algorithms are apparently really good for the dataset tested there. Although the result of the MPI_Reduce function seems to very unstable and it varies a lot during the test. This might be due to a too low number of repetitions, the very short execution time or some other factors. That the binary tree performed better than the Fibonacci tree was also quite surprising, since the communication pattern of the Fibonacci tree is almost round optimal in contrast to the binary tree.

In the second test it is clearly shown that for larger array sizes the tree algorithms perform much worse. For us the MPI_Reduce function had a better performance than our implementations when using more than 100000 array elements. This was pretty much expected after the first benchmark since for such arrays further optimizations like pipelining would be necessary.

5 Appendix

```
#include <mpi.h>
#include <string.h>
#include <stdio.h>
#include "binom_reduce.h"

void swap(void **a, void **b) {
  void *temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

int Binom_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) {

  if (root != 0) {
    fprintf(stderr, "Sorry, root!=0 not allowed");
    return -1;
  }

  int r, p, size;
  MPI_Status status;
  void *recv;
  void *reduced;

  MPI_Comm_rank(comm, &r);
  MPI_Comm_size(comm, &p);
  MPI_Type_size(datatype, &size);

  MPI_Alloc_mem(count * size, MPI_INFO_NULL, &recv);
  MPI_Alloc_mem(count * size, MPI_INFO_NULL, &reduced);

  memcpy(reduced, sendbuf, count * size);

  int i = 1;
  while ((r + i) % (2 * i) != 0 && i < p) {
    if (r + i < p) {
      MPI_Recv(recv, count, datatype, r + i, i, comm, &status);
      MPI_Reduce_local(reduced, recv, count, datatype, op);
      swap(&reduced, &recv);
    }
    i <<= 1;
  }

```

```
45   if (r != root) {
46     MPI_Send(reduced, count, datatype, r - i, i, comm);
47   } else {
48     memcpy(recvbuf, reduced, count * size);
49   }
50 
51   MPI_Free_mem(reduced);
52   MPI_Free_mem(recv);
53 
54   return 0;
55 }
```

../binom_reduce.c

```
1  #include <stdio.h>
2  #include <string.h>
3  #include <mpi.h>
4  #include "fib_reduce.h"
5 
6  int Fib_Reduce(const void *sendbuf, void *recvbuf, int count,
7      MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) {
8    if (root != 0) {
9      fprintf(stderr, "Sorry, root!=0 not allowed");
10     return -1;
11   }
12 
13   int r, p, size;
14   MPI_Status status;
15   void *recv_right;
16 
17   MPI_Comm_rank(comm, &r);
18   MPI_Comm_size(comm, &p);
19   MPI_Type_size(datatype, &size);
20 
21   int temp;
22   int right = 1;
23   int fib = 1;
24 
25   while (fib - 1 < p) {
26     temp = fib;
27     fib += right;
28     right = temp;
29   }
30 
31   int left = fib - right;
32   int i = 0;
33   int parent = 0;
```

```
34
35    while (i != r) {
36      parent = i;
37      if (r >= i + left) {
38        i += left;
39        temp = left;
40        left = right - left;
41        right = temp;
42      } else {
43        i++;
44        right -= left;
45        left -= right;
46      }
47    }
48
49    if (r == root) {
50      recv_right = recvbuf;
51    } else {
52      MPI_Alloc_mem(size * count, MPI_INFO_NULL, &recv_right);
53    }
54
55    if (right - 1 > 0 && r + 1 < p) {
56      void *recv_left;
57      MPI_Alloc_mem(count * size, MPI_INFO_NULL, &recv_left);
58      MPI_Recv(recv_left, count, datatype, r + 1, 0, comm,
     &status);
59      MPI_Reduce_local(sendbuf, recv_left, count, datatype, op);
60
61      if (left - 1 > 0 && r + left < p) {
62        MPI_Recv(recv_right, count, datatype, r + left, 0, comm,
     &status);
63        MPI_Reduce_local(recv_left, recv_right, count, datatype,
     op);
64      } else {
65        memcpy(recv_right, recv_left, count * size);
66      }
67
68      MPI_Free_mem(recv_left);
69    } else {
70      memcpy(recv_right, sendbuf, count * size);
71    }
72
73    if (r != root) {
74      MPI_Send(recv_right, count, datatype, parent, 0, comm);
75      MPI_Free_mem(recv_right);
76    }
77
```

```
  return 0;
}
```

../fib_reduce.c

```
#include <stdio.h>
#include <string.h>
#include <mpi.h>
#include "bin_reduce.h"

int int_log2(int x) {
  int r=0;
  while (x >>= 1) {
    r++;
  }
  return r;
}

int Bin_Reduce(const void *sendbuf, void *recvbuf, int count,
    MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm) {
  if (root != 0) {
    fprintf(stderr, "Sorry, root!=0 not allowed");
    return -1;
  }

  int r, p, size;
  MPI_Status status;

  MPI_Comm_rank(comm, &r);
  MPI_Comm_size(comm, &p);
  MPI_Type_size(datatype, &size);

  int tree_depth = int_log2(p) + 1;
  int i = 0;
  int depth;
  int parent = 0;

  // maximum possible number of nodes in a subtree with current
   depth
  int max_nodes = ((1 << tree_depth) - 1) / 2;

  void *recv_left;
  void *recv_right;

  if (r == root) {
    recv_right = recvbuf;
  } else {
```

```
42      MPI_Alloc_mem(count * size, MPI_INFO_NULL, &recv_right);
43    }
44
45    for (depth = 1; i != r; depth++) {
46      parent = i;
47      if (r > i + max_nodes) {
48        i += max_nodes + 1;
49      } else {
50        i++;
51      }
52      max_nodes /= 2;
53    }
54
55    if (depth != tree_depth && r + 1 < p) {
56      MPI_Alloc_mem(count * size, MPI_INFO_NULL, &recv_left);
57      MPI_Recv(recv_left, count, datatype, r + 1, 0, comm,
     &status);
58      MPI_Reduce_local(sendbuf, recv_left, count, datatype, op);
59
60      if (r + max_nodes + 1 < p) {
61        MPI_Recv(recv_right, count, datatype, r + max_nodes + 1,
     0, comm,
62            &status);
63        MPI_Reduce_local(recv_left, recv_right, count, datatype,
     op);
64      } else {
65        memcpy(recv_right, recv_left, count * size);
66      }
67
68      MPI_Free_mem(recv_left);
69    } else {
70      memcpy(recv_right, sendbuf, count * size);
71    }
72
73    if (r != root) {
74      MPI_Send(recv_right, count, datatype, parent, 0, comm);
75      MPI_Free_mem(recv_right);
76    }
77
78    return 0;
79  }
```

../bin_reduce.c