

A Specimen of Parallel Programming: Parallel Merge Sort Implementation

By Timothy J. Rolfe





1. INTRODUCTION

One common example of parallel processing is the implementation of the merge sort within a parallel processing environment. In the fully parallel model, you repeatedly split the sublists down to the point where you have single-element lists [3]. You then merge these in parallel back up the processing tree until you obtain the fully merged list at the top of the tree. While of theoretical interest, you probably don't have the massively parallel processor that this would require.

Instead, you can use a mixed strategy. Determine the number of parallel processes you can realistically obtain within your computing environment. Then construct the processing tree so that you have that number of leaf nodes. Within the leaf nodes of the processing tree, simply use the best sequential algorithm to accomplish the sorting, and send that result upstream to the internal nodes of the processing tree, which will merge the sorted sublists and then send the resulting list farther upstream in the tree. Figure One shows the processing tree for the case in which you have a list of 2000 items to be sorted and have resources only sufficient for four parallel processes. The processes receiving the size 500 lists use some sequential sorting algorithm. Because of the implementation environment, it will be something in the C/C++ language — either `qsort()` or your favorite implementation of a fast sorting algorithm.

Each leaf node (with a size 500 list) then provides the sorted result to the parent process within the processing tree. That process combines the two lists to generate a size 1000 list, and then sends that result upstream to its parent process. Finally, the root process in the processing tree merges the two lists to obtain a size 2000 list, fully sorted.

If your environment supports more parallel processes, you might take the processing tree to four levels, so that eight processes do the sequential sorting of size 250 lists. For that matter, you could even deal with circumstances in which the supported number of parallel

processes is not an exact power of two. That just means that some of the leaf nodes will be at the bottommost level and some conceptually at a higher level above in the processing tree. Since in parallel processing, the time required is the time required by the slowest process, you will probably want to stick with circumstances where the number of leaf nodes is a power of two — in other words, the processing tree is a full binary tree and all leaf nodes are doing approximately the same amount of work.



2. CHOOSING THE PARALLEL ENVIRONMENT: MPI

There is an easily used parallel processing environment for you whether your target system is a single multiprocessor computer with shared memory or a number of networked computers: the Message Passing Interface (MPI) [2]. As its name implies, processing is performed through the exchange of messages among the processes that are cooperating in the computation. As an “interface” it is not itself a full computing system, but one that is implemented in various compliant systems. Some are available without charge, such as MPICH [4] and LAM/MPI [5]. This paper will discuss the original (MPI-1) interface rather than the more recent MPI-2.

Central to computing within MPI is the concept of a “communicator”. The MPI communicator specifies a group of processes inside which communication occurs. `MPI_COMM_WORLD` is the initial communicator, containing all processes involved in the computation. Each process communicates with the others through that communicator, and has the ability to find position within the communicator and also the total number of processes in the communicator.

Through the communicator, processes have the ability to exchange messages with each other. The sender of the message specifies the process to receive the message. In addition, the sender

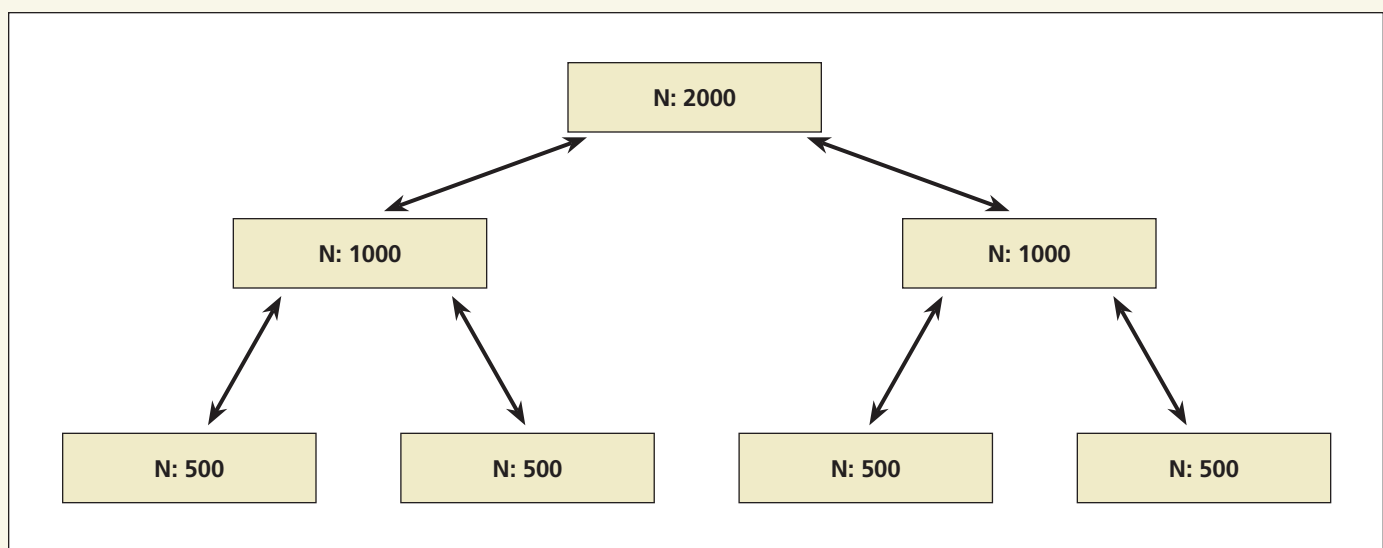


Figure 1: Three-level Sorting Tree

A Specimen of Parallel Programming: Parallel Merge Sort Implementation

continued

attaches to the message something called a message tag, an indication of the kind of message it is. Since these tags are simply non-negative integers, a large number is available to the parallel programmer, since that is the person who decides what the tags are within the parallel problem solving system being developed. Once the data buffers can safely be altered, the sending process resumes execution — in the jargon, sending is not a blocking operation.

The process receiving a message specifies both from what process it is willing to receive a message and what the message tag is. In addition, however, the receiving process has the capability of using wild cards, one specifying that it will accept a message from any sender, the other specifying that it will accept a message with any message tag. When the receiving process uses wild card specifications, MPI provides a means by which the receiving process can determine the sending process and the tag used in sending the message.

For the parallel sorting program, you can get by with just one kind of receive, the one that blocks execution until a message of the specified sender and tag is available.

MPI has many more capabilities than these, but these four, plus two more, are sufficient for the parallel sorting problem. You need to initialize within the MPI environment. The presumption is that this one is called from the program's main, and so it sends pointers to the argc and argv that it received from the operating system. The reason for this is that some implementations of MPI send information through the command-line argument vector, and so MPI needs to pull the information from it and then clean up the argument vector to reflect what the programming expects to find. The other function is the one that you use when you are finished, resigning from MPI. The six functions are these:

The first thing you do when you build an MPI application is determine the message structure: who is sending what to whom. From this framework you determine what you're going to be using as message tags. Typically you define these through #define statements so that you can use self-documenting names for the tags rather than bare numbers.

You can see from Figure One, the internal nodes in the sorting tree need to send the data to be sorted down to its child nodes. That means that some integer information must be sent — at least the size of the array segment to be sorted. Then that array segment itself needs to be sent. So there will be two messages sent downward. There is one message sent upward, the one with the sorted array segment sent to the parent process. Thus you can define three tags:

```
#define INIT 1 // Message giving size and height
#define DATA 2 // Message giving vector to sort
#define ANSW 3 // Message returning sorted vector
```

Within MPI, the cooperating processes are all started from the command line:

```
mpirun -np <number of processes> <program name>
and arguments>
```

The effect is to start all of the processes on the available computers running the specified program and receiving the indicated command-line arguments. Since all processes are running the same program, this is an example of what is called in the jargon SPMD (Single Program, Multiple Data) computation. The various processes will sort themselves and their tasks out based on their rank within the communicator. Typically you treat the rank-0 process as the privileged process, the master to which all the others are slaves. Since master/slave is distasteful to some, you can use the terminology of host process and node processes.

The host process is the one that determines the problem being solved, and then it sends subproblems down to its node processes. The node processes may themselves then communicate with other node processes. For the sorting application, the host process gets the entire vector of values to be sorted, and when the sorting is completed does whatever is required with the final result. The node processes receive their data from their parent processes within the sorting tree, send subproblems to other node processes if they are internal nodes in the sorting tree, and send their completed results back to the parent.



3. MAPPING THE COMMUNICATIONS

You might initially think of letting each node in the processing tree be a separate process. That way you can simply borrow an idea from the binary heap when it is implemented in an array with the root at zero. For any in-use cell within the array with subscript k , the left child of that heap entry is at subscript $2*k+1$, the right child is at subscript $2*k+2$, and the parent is at $(k-1)/2$. This would also give the parent/child relationships within the complete binary tree that constitutes the processing tree. Thus an internal node would split the data in half and send the two halves

TABLE 1. SET OPERATIONS

int MPI_Init(int *argc, char ***argv)	Join MPI
int MPI_Comm_rank (MPI_Comm comm, int *rank)	This process's position within the communicator
int MPI_Comm_size (MPI_Comm comm, int *size)	Total number of processes in the communicator
int MPI_Send(void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)	Send a message to process with rank dest using tag
int MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status *status)	Receive a message with the specified tag from the process with the rank source
int MPI_Finalize()	Resign from MPI

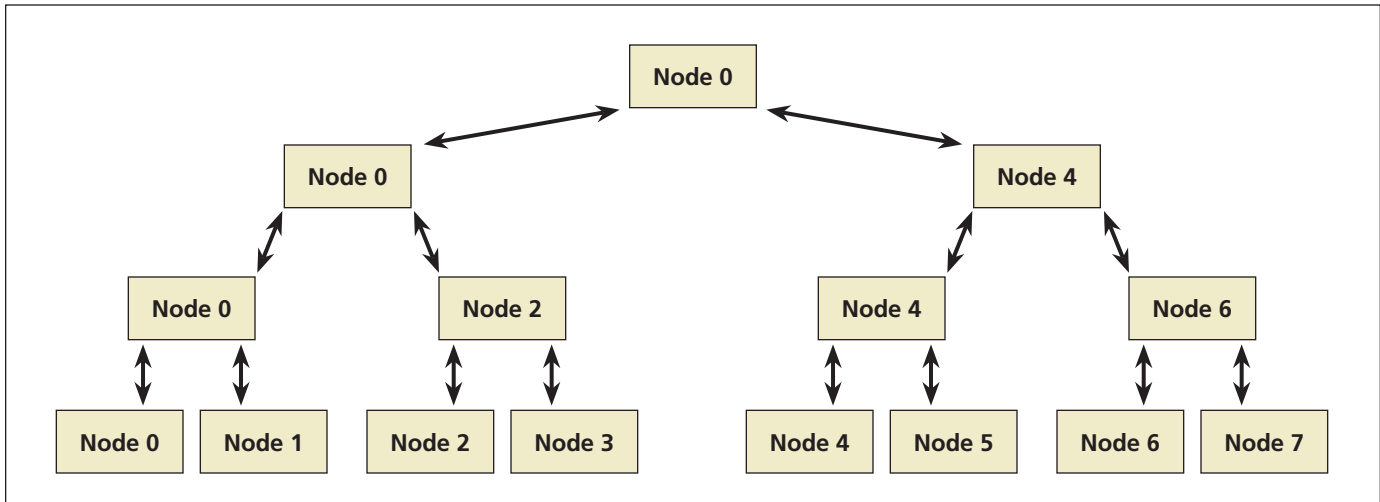


Figure 2: Process Ranks in a Four-Level Sorting Tree

to the child processes for processing. Should an internal node have only one child process, it would have to sort its own right-hand side. Leaf nodes, of course, just do the sorting. The internal nodes then receive back the data, perform the merge of the two halves, and (for all but the root node itself) send the result to the parent.

The communication of subproblems is an overhead expense that you want to minimize. Also, there's no reason to allow an internal node process to sit idle, waiting to receive two results from its children. Instead, you want the parent to send half the work to the child process and then accomplish half of the work itself. It effectively becomes a node in the next level down in the sorting tree. Figure Two shows a full sorting tree in which all of the processes (represented by their ranks) compute at the sorting tree leaf level.

Now you just need to figure out a way for a process to determine which process is its parent (from whom to receive its subproblem) and which process is its child (to whom to send a subproblem). You can do that based on the height of the node within the sorting tree, namely the number of links from this node down to a leaf node. Thus the leaf nodes are at height zero, while the initial processing of Node 0 is at height three. The multiple-level processing of the left halves can be nicely encapsulated through recursion. The processing module splits its data in half, transmits the right half subproblem to its right child, and then recursively calls itself to become the processing module for the left half within the sorting tree. (Note that this adds an additional piece of integer information to be provided to the child: which tree level the node occupies.)

Node 0 at level three needs to communicate with Node 4 — which can be computed by turning on bit 2 (assuming you number bits with bit 0 as the least-significant bit). This can be accomplished by the bit-wise operation “ $\text{myRank} | (1 \ll 2)$ ”. At level two, Node 0 needs to communicate with Node 2, while Node 4 needs to communicate with Node 6. You just

LISTING ONE: VERIFICATION OF COMMUNICATIONS SCHEMA

```
#include <stdio.h>

void communicate ( int myHeight, int myRank )
{
    int parent = myRank & ~(1<<myHeight);

    if ( myHeight > 0 )
    {
        int nxt      = myHeight - 1;
        int rtChild = myRank | ( 1 << nxt );

        printf ("%d sending data to %d\n", myRank, rtChild);
        communicate ( nxt, myRank );
        communicate ( nxt, rtChild );
        printf ("%d getting data from %d\n", myRank, rtChild);
    }
    if ( parent != myRank )
        printf ("%d transmitting to %d\n", myRank, parent);
}

int main ( void )
{
    int myHeight = 3, myRank = 0;

    printf ("Building a height %d tree\n", myHeight);
    communicate(myHeight, myRank);
    return 0;
}
```

need to turn on bit 1, or “ $\text{myRank} | (1 \ll 1)$ ”. Finally, at level one, the even nodes need to communicate with the odd nodes, turning on bit 0, or “ $\text{myRank} | (1 \ll 0)$ ”. This generalizes to “ $\text{myRank} | (1 \ll (\text{myHeight} - 1))$ ”.

Inversely, you can turn **off** the appropriate bit to get the rank for the parent node. Level 0 needs to mask off bit 0; level 1, bit 1; level 2, bit 2. Thus the height of the process determines which bit to mask off, so you complement the inverse mask and do the bit-wise AND: “ $\text{myRank} \& \sim (1 \ll \text{myHeight})$ ”. If the parent node is the same as the current node, no communication is needed, just the return from the recursive call. The left half of the array is already in place and sorted.

You can verify the communication schema by writing a tiny

A Specimen of Parallel Programming: Parallel Merge Sort Implementation

continued

```

Building a height 3 tree
0 sending data to 4
0 sending data to 2
0 sending data to 1
1 transmitting to 0
0 getting data from 1
2 sending data to 3
3 transmitting to 2
2 getting data from 3
2 transmitting to 0
0 getting data from 2
4 sending data to 6
4 sending data to 5
5 transmitting to 4
4 getting data from 5
6 sending data to 7
7 transmitting to 6
6 getting data from 7
6 transmitting to 4
4 getting data from 6
4 transmitting to 0
0 getting data from 4

```

Figure 3. Output from Communications Test

demonstration program to check it. Listing One shows such a program. It tracks just the communications and uses recursive calls both to emulate the transmission of the right halves of the arrays and the recursive calls that process the left halves.

Execution of the program (shown in Figure Three) verifies that the communications schema does track that sketched out in Figure Two. In the process, you have eliminated half of the communications required, should each node in the sorting tree be implemented as a separate process, and you have nearly cut in half the number of nodes. A full tree with root height k (using the definition of node height above) has 2^k leaf nodes and $2^k - 1$ internal nodes (since the full binary tree has $2^{k+1} - 1$ nodes in all).



4. BUILDING THE APPLICATION: SEQUENTIAL PROOF OF CONCEPT

You can implement, test, and debug the over-all scheme by developing a sequential program, using the tools available for that environment (such as single-step testing). You simply need to put in a recursive function call to replace what would be communications with the child process, but you put in place everything else that will become part of the parallel implementation.

The communications schema now becomes the prototype for the sorting program. In addition to height and rank information, the

LISTING TWO: SEQUENTIAL IMPLEMENTATION OF THE TREE SORTING ALGORITHM

```

/**
 * Partitioned merge logic
 *
 * The working core: each internal node recurses on this function
 * both for its left side and its right side, as nodes one closer to
 * the leaf level. It then merges the results into the vector passed.
 *
 * Leaf level nodes just sort the vector.
 */
void partitionedSort ( long *vector, int size, int myHeight, int myself )
{
    int parent,
        rtChild;
    int nxt;

    parent = myself & ~(1 << myHeight);
    nxt = myHeight - 1;
    rtChild = myself | ( 1 << nxt );

    if ( myHeight > 0 )
    {
        int left_size = size / 2,
            right_size = size - left_size;
        long *leftArray = (long*) calloc (left_size, sizeof *leftArray),
            *rightArray = (long*) calloc (right_size, sizeof *rightArray);
        int i, j, k; // Used in the merge logic

        memcpy (leftArray, vector, left_size*sizeof *leftArray);
        memcpy (rightArray, vector+left_size, right_size*sizeof *rightArray);

        partitionedSort ( leftArray, left_size, nxt, myself );
        partitionedSort ( rightArray, right_size, nxt, rtChild );

        // Merge the two results back into vector
        i = j = k = 0;
        while ( i < left_size && j < right_size )
            if ( leftArray[i] > rightArray[j])
                vector[k++] = rightArray[j++];
            else
                vector[k++] = leftArray[i++];
        while ( i < left_size )
            vector[k++] = leftArray[i++];
        while ( j < right_size )
            vector[k++] = rightArray[j++];
        free(leftArray); free(rightArray); // No memory leak!
    }
    else
        qsort( vector, size, sizeof *vector, compare );
}

```

module needs to receive the array that requires sorting and (since you're working in C) the number of elements in the array. If the node height is greater than zero, you compute the position of the right half of the array to generate two scratch arrays that receive the left and right portions of the array received. Instead, however, of sending the right subarray to the child process and receiving the result, you just make a recursive call on that half as well as on the left half. On the return from those two calls, do the merge of the data. If, however, the node height is zero (a leaf node), you just sort the data.

Listing Two shows `partitionedSort` — the sequential method that mimics what will become a parallel application (`parallelSort`).

LISTING THREE: MAIN() SEGMENT DISCERNING SPMD PROCESSES

```

if ( myRank == 0 )          // Host process
{ int rootHt = 0, nodeCount = 1;

  while ( nodeCount < nProc )
  { nodeCount += nodeCount; rootHt++; }

  printf ("%d processes mandates root height of %d\n",
          nProc, rootHt);
  getData (&vector, &size); // The vector to be sorted.
  // Capture time to sequentially sort an identical array
  solo = (long*) calloc ( size, sizeof *solo );
  memcpy (solo, vector, size * sizeof *solo);

  start = MPI_Wtime(); // Wall-clock time as we begin
  parallelMerge ( vector, size, rootHt);
  middle = MPI_Wtime(); // Wall-clock time after parallel sort
}
else                          // Node process
{ int  iVect[2],              // Message sent as an array
    height,                  // Pulled from iVect
    parent;                  // Computed from myRank and height
  MPI_Status status;        // required by MPI_Recv

  rc = MPI_Recv( iVect, 2, MPI_INT, MPI_ANY_SOURCE, INIT,
                MPI_COMM_WORLD, &status );
  size  = iVect[0];         // Isolate size
  height = iVect[1];        // and height
  vector = (long*) calloc (size, sizeof *vector);

  rc = MPI_Recv( vector, size, MPI_LONG, MPI_ANY_SOURCE, DATA,
                MPI_COMM_WORLD, &status );

  parallelMerge ( vector, size, height );

  MPI_Finalize();          // Resign from MPI
  return 0;                // and terminate execution.
}
// Only the rank-0 process executes here.
qsort( solo, size, sizeof *solo, compare );
finish = MPI_Wtime();      // Wall-clock time after sequential

```

* * * * *

**5. BUILDING THE APPLICATION:
PARALLEL IMPLEMENTATION**

Now you need to convert the recursive call on the right half into sending messages to the child responsible for it: send two messages to the right child. The first sends the integer information of the height of the right child in the tree and the size of the array it will be receiving. The second sends the array segment itself. You then recurse to accomplish the sorting of the left side of the array. On returning from the recursive call, accept the message from the right child giving its sorted array segment. Finally, whether leaf node or internal node, you may need to send the message to the parent with the sorted result. That occurs when the rank of the parent is different from the rank of the node itself.

Listing Three shows the SPMD portion of the main program: rank 0 gets the data and starts the ball rolling, higher ranks get their piece of the data and process that. Listing Four shows how the “par-

tionedMerge” method gets transformed into the “parallelMerge” method, as described above.

* * * * *

6. TESTING THE APPLICATION

The execution environment for this program is a Beowulf cluster comprising five 3 GHz Xeon quad-processor computers. Each Xeon processor through hyperthreading appears to the Linux operating system to have two 1.5 GHz processors, but on each machine there is a significant penalty for running more than four processes in parallel. One test of the application is to run on a single Xeon computer, forcing all processing to be done within that one machine and eliminating any message over the network. The parallel processing portion is known to have an inefficiency in that it copies data into scratch arrays that are sorted and then merged back into the original array. In addition, there is the overhead of the parallel processing itself. In the single computer test, the communication time is trivial since it amounts to exchanging messages within the same computer. The processing overhead, though, will prevent achieving the theoretical speed-up. Speed-up is the ratio of the sequential time to the parallel time, measuring elapse time (“wall clock time”) for the two. The theoretical speed-up is given by the number of processes cooperating in the calculation, which is achieved if there is no overhead in setting up the parallel processing. Figure Four shows the results of a four-process run and an eight-process run.

```

DDJ/ParallelMerge> mpirun -np 4 MPI_P_Merge
10000000
4 processes mandates root height of 2
Size: 10000000
Sorting succeeds.
  Parallel:  3.877
Sequential: 11.607
  Speed-up:  2.994
DDJ/ParallelMerge> mpirun -np 8 MPI_P_Merge
10000000
8 processes mandates root height of 3
Size: 10000000
Sorting succeeds.
  Parallel:  3.643
Sequential: 11.573
  Speed-up:  3.177

```

Figure 4. Single Computer Results

For comparison, you can look at the timing results for comparable processing trees when you have separate nodes for the internal nodes of the tree, thus requiring twice the communications. Those

A Specimen of Parallel Programming: Parallel Merge Sort Implementation

*continued***LISTING FOUR: PARALLELMERGE PROCEDURE**

```

void parallelMerge ( long *vector, int size, int myHeight )
{
    int parent;
    int myRank, nProc;
    int rc, nxt, rtChild;

    rc = MPI_Comm_rank (MPI_COMM_WORLD, &myRank);
    rc = MPI_Comm_size (MPI_COMM_WORLD, &nProc);

    parent = myRank & ~(1 << myHeight);
    nxt = myHeight - 1;
    rtChild = myRank | ( 1 << nxt );

    if ( myHeight > 0 )
    { //Possibly a half-full node in the processing tree
        if ( rtChild >= nProc ) // No right child; down one level
            parallelMerge ( vector, size, nxt );
        else
        {
            int left_size = size / 2,
                right_size = size - left_size;
            long *leftArray = (long*) calloc (left_size,
                sizeof *leftArray),
                *rightArray = (long*) calloc (right_size,
                sizeof *rightArray);

            int iVect[2];
            int i, j, k; // Used in the merge logic
            MPI_Status status; // Return status from MPI

            memcpy (leftArray, vector,
                left_size*sizeof *leftArray);
            memcpy (rightArray, vector+left_size,
                right_size*sizeof *rightArray);
            iVect[0] = right_size;
            iVect[1] = nxt;
            rc = MPI_Send( iVect, 2, MPI_INT, rtChild, INIT,
                MPI_COMM_WORLD);
            rc = MPI_Send( rightArray, right_size, MPI_LONG, rtChild,
                DATA, MPI_COMM_WORLD);

            parallelMerge ( leftArray, left_size, nxt );
            rc = MPI_Recv( rightArray, right_size, MPI_LONG, rtChild,
                ANSW, MPI_COMM_WORLD, &status );

            // Merge the two results back into vector
            i = j = k = 0;
            while ( i < left_size && j < right_size )
                if ( leftArray[i] > rightArray[j])
                    vector[k++] = rightArray[j++];
                else
                    vector[k++] = leftArray[i++];
            while ( i < left_size )
                vector[k++] = leftArray[i++];
            while ( j < right_size )
                vector[k++] = rightArray[j++];
        }
    }
    else
        qsort( vector, size, sizeof *vector, compare );

    if ( parent != myRank )
        rc = MPI_Send( vector, size, MPI_LONG, parent, ANSW,
            MPI_COMM_WORLD );
}

```

results are shown in Figure Five.

```

DDJ/ParallelMerge> mpirun -np 7 MPI_T_Merge
10000000
Size: 10000000
Sorting succeeds.
Parallel: 4.191
Sequential: 11.452
Speed-up: 2.733
DDJ/ParallelMerge> mpirun -np 15 MPI_T_Merge
10000000
Size: 10000000
Sorting succeeds.
Parallel: 3.907
Sequential: 11.492
Speed-up: 2.941

```

Figure 5. Alternative Implementation, More Messages

On the other hand, you can force network communications by running the application in an MPI session involving multiple computers, and letting MPI think that each computer has only one processor. In this environment, the mismatch between processor speed and network communications speed becomes obvious. Each Xeon processor is a 3 GHz hyperthreaded processor, so that Linux sees two 1.5 GHz processors. A 100 Mbit network connects the five computers — quite slow as compared with processing speed. Figure Five shows the results in this environment — significantly worse than the single-computer results.

```

DDJ/ParallelMerge> mpirun -np 4 MPI_P_Merge
10000000
4 processes mandates root height of 2
Size: 10000000
Sorting succeeds.
Parallel: 8.421
Sequential: 11.611
Speed-up: 1.379
DDJ/ParallelMerge> mpirun -np 8 MPI_P_Merge
10000000
8 processes mandates root height of 3
Size: 10000000
Sorting succeeds.
Parallel: 8.168
Sequential: 11.879
Speed-up: 1.454

```

Figure 6. Networked Computer Results

Any network communications drastically degrades the performance. You can try letting MPI know that each computer has four processors. In that case, MPI will deal out processes by fours before it moves to the next available computer. Thus, if you ask for eight parallel processes, ranks 0 through 3 will be on one computer, with fastest possible communications, and ranks 4 through 7 will be on another computer. Thus the only mes-

saging is at the root, when rank 0 sends its right half to rank 4. Figure Seven shows those results — a speed-up of 3.177 comes down to a speed-up of 2.025. Because of the “verbose” flag to `mpirun`, you can see the processes starting by rank. It is, however, better than the 1.454 that you get when *all* communications are over the network.

```
DDJ/ParallelMerge> mpirun -v -np 8 MPI_P_Merge
10000000
28823 MPI_P_Merge running on n0 (o)
28824 MPI_P_Merge running on n0 (o)
28825 MPI_P_Merge running on n0 (o)
28826 MPI_P_Merge running on n0 (o)
614 MPI_P_Merge running on n1
615 MPI_P_Merge running on n1
616 MPI_P_Merge running on n1
617 MPI_P_Merge running on n1
8 processes mandates root height of 3
Size: 10000000
Sorting succeeds.
Parallel: 5.705
Sequential: 11.555
Speed-up: 2.025
```

Figure 7. Results from Networked SMP Computers



7. CLOSING COMMENTS

It is not too *terribly* painful to take a sequential algorithm and split it apart into components that can run in parallel. If, however, there is significant message passing, the improvements promised by parallel processing can be greatly diminished by a mismatch between processing speed on each computer and the communications time for messages exchanged between them.

The URL <http://penguin.ewu.edu/~trolfe/ParallelMerge/> provides access to the complete programs represented by the listings are available with their supporting main and other methods. In ad-

dition, it includes an implementation with PVM (Parallel Virtual Machine), [1] an earlier public-domain method of developing distributed processing programs, as well as the MPI implementation in which all of the internal nodes are separate processes.

The author developed this program as part of teaching the Eastern Washington University course CSCD-543, Distributed Multiprocessing Environments, in the Winter 2006 quarter. (Information about that class is available at <http://penguin.ewu.edu/class/cscd543/>.) The computations were performed on the computers acquired as part of the “Technology Initiative for the New Economy” (TINE) Congressional grant to Eastern Washington University that, among other things, provided a parallel and distributed processing resource — which these computers do admirably well! **lr**

References

- [1] Geist, Al, Adam Beguelin, Jack Dongarra, Weicheng Jiang, Robert Manchek, and Vaidia Sundaram, *PVM: Parallel Virtual Machine — A Users' Guide and Tutorial for Networked Parallel Computing* (The MIT Press: 1997). On the web, see <http://www.csm.ornl.gov/pvm/>
- [2] Gropp, William, Ewing Lusk, and Anthony Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface* (The MIT Press: 1999). On the web, see <http://www.mcs.anl.gov/mpil/>.
- [3] Roosta, Sayed, H., (for example) *Parallel Processing and Parallel Algorithms* (Springer-Verlag New York: 2000), pp. 397-98.
- [4] <http://www.mcs.anl.gov/research/projects/mpich2/> Note that this implements both the MPI-1 and the MPI-2 standards.
- [5] <http://www.lam-mpi.org/>

TIMOTHY J. ROLFE

Computer Science Department
Eastern Washington University
319F Computing & Engineering Building
Cheney, Washington 99004-2493 USA
TRolfe@ewu.edu

Categories and Subject Descriptors: D.1.3 [Software]: Programming Techniques - Concurrent Programming (Distributed programming)

General terms: Algorithms, Performance, Experimentation

Keywords: algorithm transformation, MPI, merge sort

DOI: 10.1145/1869746.1869767

© 2010 ACM 2153-2184/10/1200 \$10.00

ACM-W

ACM's Committee on Women in Computing



NEWS/PUBLICATIONS • PROJECTS • AMBASSADORS

INTERNSHIPS • RELATED SITES • RESEARCH



<http://www.acm.org/women/>