An Evaluation of The Message-Passing Interface

PER BRINCH HANSEN

Syracuse University, 2-120 CST, Syracuse, NY 13244

pbh@top.cis.syr.edu

December 1997

Abstract: The Message-Passing Interface (MPI) is evaluated by rewriting message parallel programs for Householder reduction, matrix multiplication, and successive overrelaxation. The author concludes that MPI is a practical programming tool. It does, however, lack the elegance and security that can only be achieved by a parallel programming language.

Keywords: programming languages; parallel programming; message parallelism; The Message-Passing Interface

INTRODUCTION

The Message-Passing Interface (MPI) defines a standard library of subroutines for message parallelism in portable programs written in C or Fortran [1]. The purpose of the present work is to evaluate MPI as a programming tool and discover the pitfalls of replacing the abstract notation of a parallel programming language by a library of subroutines.

The starting point of this evaluation was a collection of model programs for parallel scientific computing [2–5]. These programs were originally written in SuperPascal [6] and occam [7] for message parallel architectures, such as the Meiko Computing Surface [8].

Each model program is viewed as an instance of a *programming paradigm*—a class of programs that solve different problems, but have the same control structure. A *model program* has a parallel component that implements the common control structure and a sequential component for a specific application. The clear separation of the issues of parallelism and the details of application leads to programs that are easy to understand.

The common control structure of a programming paradigm is sometimes called an algorithmic skeleton, a generic program, or a program template. This concept has also been explored by Cole [9], Dongarra [10], and others.

The experiments described in the following were planned by me and carried out by Vančo Burzewski. We evaluated MPI by rewriting three of the model programs in C for an IBM SP-2 system with 12 nodes. This paper concentrates on the essence of MPI and ignores minor details.

THE HOUSEHOLDER PIPELINE

We began by rewriting a SuperPascal program that uses Householder reduction to solve a linear system $ax = b$. The program uses parallel processors which communicate by unbuffered messages only. A master processor sends the matrix a and the vector b through a pipeline, which reduces the matrix to triangular form. The master then receives the reduced matrix and computes the solution vector x by backsubstitution.

The following program fragment, written in SuperPascal, defines how a pipeline node receives one column a_j at a time from its left neighbor and sends it to its right neighbor:

```
for j := r - 1 downto 1 do
  begin
    receive(left, aj);
    send(right, aj);
  end
```

The standard procedures, *send* and *receive*, are part of SuperPascal. Consequently, the compiler can use type analysis to determine the length of a message and include it as a hidden parameter in the code.

In MPI, the same piece can be expressed as follows:

```
for (j = r - 1; j >= 1; j--)
  { MPI_Recv(aj, n + 1, MPI_DOUBLE, left,
      0, MPI_COMM_WORLD, &Status);
    MPI_Send(aj, n + 1, MPI_DOUBLE, right,
      0, MPI_COMM_WORLD);
  }
```

Since C does not support parallelism directly, message passing must be handled by library procedures. And these procedures must be general enough to handle all possible communications without any help from a compiler. As a result, the communication procedures now require 6–7 (instead of 2) parameters.

If you take professional pride in writing concise programs, such notational clutter is unacceptable. Fortunately, there is a simple remedy for this problem. In the Householder pipeline, all messages are column vectors of the same length. For this special case, we defined a macro for sending a vector x of length $n+1$ to a destination identified by a processor number:

```
#define send_vector(destination, x) \
  MPI_Send(x, n + 1, MPI_DOUBLE, destination, \
    0, MPI_COMM_WORLD)
```

A similar macro was introduced for receiving a vector, where

```
typedef double vector[n+1];
```

The vector elements are numbered 1, 2, ..., n. (Element number 0 is unused).

Using these macros, the MPI algorithm for a pipeline node (Algorithm 1) is very close to the SuperPascal version [5].

```
void node(int r, int s, int left, int right)
{ typedef vector block[q + 1];
  block a, v; vector aj, b; int i, j;
  receive_vector(left, b);
  for (i = 0; i <= s - r; i++)
    { receive_vector(left, a[i]);
      for (j = 0; j <= i - 1; j++)
        transform(j + r, a[i], v[j]);
      eliminate(i + r, a[i], v[i]);
      transform(i + r, b, v[i]);
    };
  send_vector(right, b);
  for (j = s + 1; j <= n; j++)
    { receive_vector(left, aj);
      for (i = 0; i <= s - r; i++)
        transform(i + r, aj, v[i]);
      send_vector(right, aj);
    };
  for (i = s - r; i >= 0; i--)
    send_vector(right, a[i]);
  for (j = r - 1; j >= 1; j--)
    { receive_vector(left, aj);
      send_vector(right, aj);
    }
}
```

Algorithm 1. Householder node.

The pipeline consists of p processors. From its left neighbor, a node inputs vector b followed by columns a_r through a_s, where $1 \le r \le s \le n-1$. These columns are stored and transformed in a local matrix a which holds q (or $q+1$) columns, where $q = (n-1)/p$. The remaining columns, a_{s+1} through a_n, are then input from the left neighbor, transformed, and output to the right neighbor. Finally, the node outputs its own (reduced) columns in reverse order,

$$a_s, a_{s-1}, \ldots, a_r$$

and copies columns

$$a_{r-1}, a_{r-2}, \ldots, a_1$$

output in reverse order by the previous nodes. The node algorithm is explained in detail in [5].

The master processor initializes and outputs vector b and matrix a, before inputting the reduced matrix a. Finally, it computes the solution vector x by backsubstitution and displays it (Algorithm 2).

```
void master(int left, int right)
{ typedef vector matrix[n + 1];
  matrix a; vector b, x; int i;
  initialize(a, b);
  send_vector(left, b);
  for (i = 1; i <= n; i++)
    send_vector(left, a[i]);
  receive_vector(right, b);
  for (i = n; i >= 1; i--)
    receive_vector(right, a[i]);
  substitute(a, b, x);
  display(x);
}
```

Algorithm 2. Householder master.

The master and the pipeline form a ring network. Initially, all processors execute the main procedure of the program (Algorithm 3). Each processor has an ordinal number (or *rank*) k. Processor 0 is the master. The pipeline consists of processors 1, 2, ..., p. An ellipsis, ..., marks the omission of minor details.

```
void main(...)
{ int k, rem;
  MPI_Init(...);
  MPI_Comm_rank(..., &k, ...);
  if (k == 0) master(1, p);
  else /* 1 <= k <= p */
    { rem = (n - 1)%p;
      if (k <= rem)
        /* q + 1 columns per node */
        node((k - 1)*(q + 1) + 1, k*(q +1),
          k - 1, (k + 1)%(p + 1));
      else
        /* q columns per node */
        node((k - 1)*q + rem + 1, k*q + rem,
          k - 1, (k + 1)%(p + 1));
    };
  MPI_Finalize();
}
```

Algorithm 3. Main Householder block.

Table I shows the measured (and predicted) run times T_p of this program on an IBM SP-2 system for the Householder pipeline with p processors. The parallel run time is

$$T_p = a(1 + f)n^3/p + bpn^2$$

The factor $f = (1 - 1/p)(2 - 1/p)$ is derived in [5]. For the SP-2 system, the constants of computation and communication are $a = 0.15$ μs and $b = 10$ μs, respectively.

The Householder pipeline described here distributes the computational load very unevenly among the processor nodes. We chose it only to experiment with the MPI notation and find out if parallel performance can be predicted accurately for the SP-2 implementation.

Table I Householder reduction.

n	p	$T_p(s)$	E_p
1200	1	265 (274)	1.00
1200	2	262 (256)	0.51
1200	3	226 (226)	0.39
1200	4	204 (207)	0.32

THE MULTIPLICATION PIPELINE

Next, we evaluated the potential efficiency of MPI programs by rewriting a SuperPascal program for multiplication of two $n \times n$ real matrices [5]. The program defines a pipeline which divides the computational load evenly among p processors.

```
void node(int r, int s, int left, int right)
{ typedef vector block[q + 1];
  block, a, c; int i, j;
  vector ai, bj, ci;
  for (i = 0; i <= s - r; i++)
    receive_vector(left, a[i]);
  for (i = s + 1; i <= n; ++i)
    { receive_vector(left, ai);
      send_vector(right, ai);
    };
  for (j = 1; j <= n; j++)
    { receive_vector(left, bj);
      if (s < n) send_vector(right, bj);
      for (i = 0; i <= s - r; i++)
        c[i][j] = f(a[i], bj);
    };
  for (i = 1; i <= r - 1; i++)
    { receive_vector(left, ci);
      send_vector(right, ci);
    };
  for (i = 0; i <= s - r; i++)
    send_vector(right, c[i]);
}
```

Algorithm 4. Multiplication node.

Algorithm 4 defines the behavior of a pipeline node. First, the n rows of a square matrix a are input and distributed evenly among the nodes of the pipeline. Then the columns of a matrix b pass through the pipeline, while each node computes a portion of the product matrix $c = a \times b$. Finally, the product matrix is output by the pipeline. The algorithm assumes that a and c are stored by rows, while b is stored by columns. The function f computes the dot product of a row a_i and a column b_j.

Table II shows the measured (and predicted) run times T_p of this program on an IBM SP-2 system for the multiplication pipeline with p processors.

Table II Matrix multiplication.

n	p	T_p (s)	E_p
840	1	165 (166)	1.00
840	3	107 (102)	0.51
840	5	116 (118)	0.28
840	7	141 (145)	0.17

The parallel run time is

$$T_p = an^3/p + bpn^2$$

On the SP-2 system, communication is two orders of magnitude slower than computation: $a = 0.25$ μs and b = 25 μs. Furthermore, the communication time is proportional to the number of processors p. This apparently means that all communications in the SP-2 system take place one at a time. Consequently, the processor efficiency, $E_p = T_1/(pT_p)$, is remarkably poor. With seven processors, the program runs only 17% faster than it does on a single processor.

By contrast, for $n = 800$, the original program written in *occam* achieved a parallel speedup of 18 on a Meiko Computing Surface with 20 transputers [5].

THE LAPLACE MATRIX

Finally, we rewrote a SuperPascal program that solves Laplace's heat equation for a square real matrix using successive overrelaxation [4]. The program runs on a square processor matrix with p nodes.

Algorithm 5 defines the identical behavior of the nodes.

```
void node(int qi, int qj, int steps, int up,
  int down, int left, int right)
{ subgrid u; int k;
  fopt = 2.0 - 2.0*pi/n;
  newgrid(qi, qj, u);
  for (k = 1; k <= steps; k++)
    relax(qi, qj, up, down, left, right, u);
  output(qi, qj, right, left, u);
}
```

Algorithm 5. Laplace node.

Each node is identified by its row and column numbers (q_i, q_j) in the processor matrix:

$$1 \le q_i \le q, \qquad 1 \le q_j \le q, \qquad \text{where} \qquad q = \sqrt{p}$$

A node communicates only with its four nearest neighbors (if any). The neighboring nodes are identified by ordinal processor numbers, *up*, *down*, *left*, and *right*, in the range from 0 to $p - 1$.

The entire $n \times n$ grid of temperatures is distributed evenly among the $q \times q$ nodes. Each node generates a subgrid of $m \times m$ interior temperatures surrounded by a boundary of temperatures received from neighboring nodes, where $m = n/q$:

```
typedef double vector[m + 2];
typedef vector subgrid[m + 2];
```

A node updates (or *relaxes*) its subgrid a fixed number of times before outputting final temperatures.

In each relaxation step, a node exchanges boundary values with its neighbors (Algorithm 6). The programming details are explained in [4].

Now, if the nodes have identical behavior, the synchronous communication may cause deadlock. This is avoided by defining two kinds of nodes which alternate like black and white squares on a chessboard (Algorithm 7).

A black node sends a vector to each of its neighbors (if any), and receives a vector from each neighbor (Algorithm 8). Every send operation performed by a black node is matched by a corresponding receive operation performed by a white node (and vice versa). Consequently, the communication sequence for a white node is very similar (Algorithm 9).

The procedure *send_column* copies column u_j of a subgrid into a local vector, x, and sends it to a given destination:

```
void send_column(int destination, subgrid u, int j)
{ vector x; int i;
  for (i = 0; i <= m + 1; i++)
    x[i] = u[i][j];
  send_vector(destination, x);
}
```

```
void relax(int qi, int qj, int up, int down,
  int left, int right, subgrid u)
{ int b, i, j, k;
  for (b = 0; b <= 1; b++)
    { exchange(qi, qj, 1 - b, up, down,
        left, right, u);
      for (i = 1; i <= m; i++)
        { k = (i + b)%2;
          j = 2 - k;
          while (j <= m - k)
            { nextstate(u, i, j);
              j = j + 2;
            }
        };
    }
}
```

Algorithm 6. Subgrid relaxation.

```
void exchange(int qi, int qj, int b, int up,
  int down, int left, int right, subgrid u)
{ if ((qi + qj)%2 == 0)
    blacknode(qi, qj, b, up, down, left, right, u);
  else
    whitenode(qi, qj, b, up, down, left, right, u);
}
```

Algorithm 7. Data exchange.

```
void blacknode(int qi, int qj, int b, int up,
  int down, int left, int right, subgrid u)
{ if (qi > 1)
    { send_row(up, u, 1);
      receive_row(up, u, 0);
    };
  if (qj < q)
    { send_column(right, u, m);
      receive_column(right, u, m + 1);
    };
  if (qi < q)
    { send_row(down, u, m);
      receive_row(down, u, m + 1);
    };
  if (qj > 1)
    { send_column(left, u, 1);
      receive_column(left, u, 0);
    }
}
```

Algorithm 8. Black node communication.

```
void whitenode(int qi, int qj, int b, int up,
  int down, int left, int right, subgrid u)
{ if (qi < q)
    { receive_row(down, u, m + 1);
      send_row(down, u, m);
    };
  if (qj > 1)
    { receive_column(left, u, 0);
      send_column(left, u, 1);
    };
  if (qi > 1)
    { receive_row(up, u, 0);
      send_row(up, u, 1);
    };
  if (qj < q)
    { receive_column(right, u, m + 1);
      send_column(right, u, m);
    }
}
```

Algorithm 9. White node communication.

The procedures *receive_column*, *send_row*, and *receive_row* are very similar.

These communication procedures use a macro to send a column (or row) vector of a subgrid. It is derived from the macro, *send_vector*, of the Householder pipeline by replacing the vector length $n + 1$ by $m + 2$. A similar macro is used to receive a vector.

Table III shows measured run times, T_p, in seconds for 800 relaxations of an 800×800 grid on 1 and 4 processors. Even for such a small number of processors, the parallel efficiency, $E_4 = T_1/(4T_4)$, is only 62% for a computational problem that is ideally suited for parallelism.

Table III Successive overrelaxation.

n	p	T_p (s)	E_p
800	1	408	1.00
800	4	165	0.62

The same program, written in *occam*, relaxed a 750×750 grid 750 times on 9 transputers with a processor efficiency of 97% [4].

CONCLUSIONS

The *Message-Passing Interface* (MPI) has been evaluated by rewriting parallel model programs for matrix multiplication, Householder reduction, and successive overrelaxation in C for an IBM SP-2 system. Here is what I learned:

1. A well-structured program written in a message parallel language, such as SuperPascal or occam, can be rewritten in MPI in a straightforward manner.

2. The decision to replace the simple standard procedures of a well-designed programming language by complicated library procedures with long parameter lists clutters the notation. This complexity can, however, be hidden by introducing send and receive macros with short parameter lists for each type of message used in a program.

3. For standard matrix computations, the substantial overhead of message communication implemented by subroutines (instead of machine instructions) effectively limits the use of MPI to parallel architectures with, say, ten nodes or less. (However, serialized message switching in the IBM SP-2 system is an even more serious performance limitation.)

4. The MPI routines for synchronous message passing work as expected. However, asynchronous communication is dangerously insecure. It is possible to call a user procedure that inputs a message in a local variable and returns

before the input has been completed. This time-dependent error may change a variable, which (conceptually) no longer exists, and therefore may be reused by unrelated procedure calls!

Twenty years ago, Concurrent Pascal proved that nontrivial parallel programs can be written exclusively in a secure programming language [11].

The Message-Passing Interface follows in the footsteps of the Unix threads library: both extend a sequential programming language with subroutines for parallel execution and data communication.

Personally, I regard the attempt to replace a parallel programming language and its compiler with insecure procedures as a step backwards in programming technology.

Acknowledgements

I appreciate the comments of David Jakel and Stephen Taylor. This work was supported by the National Science Foundation under grant number CCR-9311759. It was conducted using the computational resources of the Northeast Parallel Architectures Center (NPAC) at Syracuse University.

References

1. Gropp, W., Lusk, E. and Skjellum, A. 1994. *Using MPI*, Cambridge MA: The MIT Press.
2. Brinch Hansen, P. 1992. Householder reduction of linear equations. *ACM Computing Surveys* 24: 185–194.
3. Brinch Hansen, P. 1993a. Model programs for computational science: a programming methodology for multicomputers. *Concurrency—Practice and Experience* 5: 407–423.
4. Brinch Hansen, P. 1993b. Parallel cellular automata: a model program for computational science. *Concurrency—Practice and Experience* 5: 425–448.
5. Brinch Hansen, P. 1995. *Studies in Computational Science: Parallel Programming Paradigms.* Englewood Cliffs NJ: Prentice Hall.
6. Brinch Hansen, P. 1994. The programming language SuperPascal. *Software—Practice and Experience* 24: 467–483.
7. Inmos Ltd. 1998. *occam 2 Reference Manual*, Englewood Cliffs NJ: Prentice Hall.
8. McDonald, N. 1991. Meiko Scientific Ltd. In *Past. Present, Parallel: A Survey Of Available Parallel Computing Systems*, A. Trew and G. Wilson eds., New York: Springer-Verlag, 165–175.
9. Cole, M. I. 1989. *Algorithmic Skeletons: Structured Management of Parallel Computation.* Cambridge MA: The MIT Press.
10. Dongarra, J. J. and Sorenson, D. C. 1989. Algorithmic designs for different computer architectures. In *Opportunities and Constraints of Parallel Computing*, J. L. C. Sanz ed., New York: Springer Verlag, 33–35.
11. Brinch Hansen, P. 1975. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering* 1: 199–207.