Extended Precision Multiplication using a Message Passing Interface (MPI)*

Bill McDaniel[1] and Evan Lemley[2]
[1]Department of Computer Science
University of Central Oklahoma
Edmond, OK 73034
billmcd@roesner.us
[2]Center for Research and Education in Interdisciplinary Computation
College of Mathematics and Science
University of Central Oklahoma
Edmond, OK 73034
elemley@uco.edu

Abstract

This tutorial describes the Trachtenberg System **[trachtenberg]** of multiplication using a Message Passing Interface (MPI) **[mpi40]** running on multiple processors. Multiplication on a single processor is given first, then multiplication using multiple processors follows.

1 Single Processor Implementation

The essential logic of the non-MPI code is given as:

1. accept 2 strings (all digits)
2. make sure that they are the same length
3. convert from characters to numbers
4. do the multiplication

5. return the answer (as a string)

The original version of the multiplication routines were written as 2 functions in C++, and is shown below.

Listing 1: This is the main multiplication routine

```
string multiply(string a, string b)
 {
   int j,k, ls,rs;
   string ans = "";

   // make the strings equal length, jic
   while (a.size() != b.size())
     if (a.size() < b.size()) a = '0'+a; else b = '0'+b;

   // convert from characters to numbers so the
   // multiplications will work correctly
   int numdigits = a.size();
   for (int i=0; i<numdigits; i++) { a[i]-='0';b[i]-='0';}

   // do the multiply - 'ls' is the position of the
   // left side of the group to be multiplied,
   // and 'rs' is the position of the right side
   ls = rs = numdigits-1;
   int carry=0;
   do
     {
       for (k=ls, j=rs; k<=rs; k++,j--) carry += a[j]*b[k];
       handle_carries(carry,ans);
       if (ls > 0) ls--; else rs--;
     }
   while (rs > -1);

   // take care of any extra values in 'carry'
   while (carry) handle_carries(carry,ans);
   return ans;
 }
```

Listing 2: The following function converts the rightmost digit to a character and prepends it onto the answer then strips the rightmost digit

```
void handle_carries(long long int & carry, string & ans)
  {
    ans = char(carry%10+'0') + ans;
    carry /= 10;
  }
```

Notes:

1. The variable *ans* will contain the product when the algorithm is finished.
2. The *handle_carries* routine will pick off the rightmost digit, convert it to a character, then prepend it to the answer, creating the answer right to left.
3. A machine readable version of the code can be found at www.roesner.us/ billmcd
4. There may be a question about the maximum value for *carry*. Testing two 3000 digit numbers, we found that the maximum number for *carry* is 269990, so overflow is not a major concern.

This particular system for multiplication is not widely known, so, a specific example is shown below in 1. The reader will note that the answer is created in reverse order (right to left).

2 Multiple Processor Implementation

Converting the above algorithm over to run using multiple processors is fairly straightforward. The approach used was to have one processor for each output digit.

The objective of this implementation was to have the calculations for various instances of t done at the same time, on different processors.

The MPI version of this code was run on the Buddy supercomputer at the University of Central Oklahoma. This resource was funded by a National Science Foundation grant (OAC 1429702) and consists of 31 general purpose nodes each with two ten-core Intel Xeon CPUs and 64 GB of RAM, and four high-memory nodes (identical to the general purpose nodes except for RAM size of 128 GB), and two GPU nodes each with one NVidia Tesla K-40 card.

Message Passing Interface (MPI) was introduced as a method of paralleliz-ing code in 1994 and has continued to be one of the primary ways code is run in parallel on high performance computing platforms.

The MPI functions that are used in the program are shown in 2. Default values are used whenever possible.

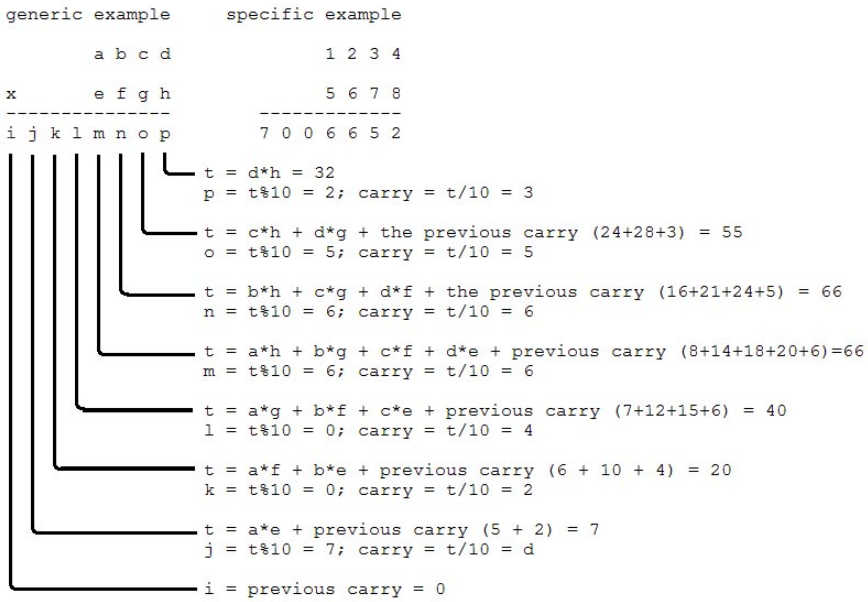

Figure 1: Given two 4 digit numbers that are to be multiplied together, where the digits are represented by letters, p is determined first, then o, then n, and so on. The variable t is a temporary variable to hold intermediate values.

```
The following MPI functions are used in the program.  Default values
are used whenever possible.

    // Initialize the MPI environment
    MPI_Init(NULL, NULL);

    // Get the number of processes
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);
    //                            └── a locally defined integer

    // Get the rank of the process
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    //                            └── a locally defined  integer
    // The variable 'my_rank' is a locally defined integer that is used
    // to identify each process.

    // send an integer
    MPI_Send(&rs,  1, MPI_INT, i, 0,  MPI_COMM_WORLD);
    //                         └── 'to' process
    //                      └── type of data being sent
    //                   └── number of elements
    //            └── the data sent

    // receive an integer
    MPI_Recv(&digit, 1, MPI_INT,   i, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    //                             └── 'from' process
    //                          └── type of date being received
    //                       └── number of elements
    //            └── the data received

    // this function terminates the MPI execution environment
    MPI_Finalize();
```

Figure 2: MPI functions used in the multiplication program

The program is basically divided into 3 parts.

1. if $world\_rank = 0$ then send the endpoints of the various groups to each processor and wait for the results to be returned.
2. if $(world\_rank > 0 \;\&\&\; world\_rank <= N * 2 - 1)$ then accept ls and rs and do the multiplications.
3. if $(world\_rank = N * 2)$ then take care of the leftmost digit.

A quick summary of the logic of the program follows:

- Get the number of processes
- Get the rank of the process
- Get the name of the processor

pseudocode for process 0

```
determine the value of the left side and the right side
send the appropriate values of ls and rs to each process
    for (i=1; i <= N*2-1;i++)
        send ls to process i
        send rs to process i
        send a value of 0 (carry) to process 0
        if (ls > 0) ls--; else rs--;
receive the digits back (right to left)
    for (i=1; i <= N*2; i++)
        receive 1 digit
        result [N*2-i]= digit;
print the result - skip leading zeros
```

pseudocode for processes $= 1...N*2-1$

```
get ls and rs from ps 0
do the computations
    for (k=ls, j=rs; k <= rs; k++,j--) carry += a[j]*b[k];
read tcarry from the previous process then add tcarry to carry
pick off the rightmost digit and send to ps 0
compute the new value for carry and send to the next process
    carry = carry % 10
    send carry to the next process
code for process N*2
    get the carry from the previous process
    send the carry to ps 0
Finalize the MPI environment
```

Since the number of processes required has to be determined before the program starts, a shell script is used to determine some of the run time constants.

Listing 3: bash script to determine number of processes and other run-time constants before actual run begins

```
#!/bin/bash
#
echo "entering multiply.sh"
u=$1
v=$2
```

```
if [[ $u && $v ]]
then
  # determine the lengths of the 2 parameters
  lu=${#u}
  lv=${#v}

  # set lm to the length of the longer string
  if [ ${lu} -gt ${lv} ]; then lm=$lu; else lm=$lv; fi
  echo "␣-length␣of␣the␣longest␣string␣=␣"$lm

  # calculate the number of tasks needed
  p=$((lm*2+1))
  echo "␣-number␣of␣tasks␣"$p

  echo "␣-compiling"
  mpicc multiply.c

  # exit on compile time error
  if [ $? != 0 ]; then exit; fi

  echo "run␣the␣program"
  mpirun -np $p  "a.out" $u  $v   $lm
else
  echo  "usage:␣bash␣multiply.sh␣number␣number"
fi
```

Students in a senior level parallel programming course (or related) could be first asked to develop a single processor algorithm for Trachtenberg multiplication, then asked to develop a multiprocessor version of the single-processor code.

3 Acknowledgements

The code and scripts in this tutorial were developed and tested on the Buddy Supercomputer at the University of Central Oklahoma, which was funded by National Science Foundation grant, OAC-1429702.