Engineering Physics and Mathematics Division

Mathematical Sciences Section

**SYNTHETIC MODELS OF DISTRIBUTED
MEMORY PARALLEL PROGRAMS**

David A. Poplawski

Department of Computer Science
Michigan Technological University
Houghton, MI 49931

Date Published: September, 1990

Research was supported by the
Applied Mathematical Sciences Research Program
of the Office of Energy Research,
U.S. Department of Energy.

Prepared by the
Oak Ridge National Laboratory
Oak Ridge, Tennessee 37831
operated by
Martin Marietta Energy Systems, Inc.
for the
U.S. DEPARTMENT OF ENERGY
under Contract No. DE-AC-05-84OR21400

MASTER

20

Contents

1 Introduction	1
2 Definitions	2
3 Creating synthetic benchmarks	2
3.1 From scratch	3
3.2 From an existing program	5
3.3 From observed behavior	7
4 Performance prediction using synthetic benchmarks	12
5 Summary	13
5.1 Further work	14
Acknowledgements	14
References	15
A Source code for Gaussian elimination with pivoting	16
B Source code for wavefront algorithm	23
C Graphs of message types and lengths for wavefront algorithm	24

SYNTHETIC MODELS OF DISTRIBUTED MEMORY PARALLEL PROGRAMS

David A. Poplawski

Abstract

This paper deals with the construction and use of simple synthetic programs that model the behavior of more complex, real parallel programs. Synthetic programs can be used in many ways: to construct an easily ported suite of benchmark programs, to experiment with alternate parallel implementations of a program without actually writing them, and to predict the behavior and performance of an algorithm on a new or hypothetical machine. Synthetic programs are constructed easily from scratch, from existing programs, and can even be constructed using nothing but information obtained from traces of the real program's execution.

1. Introduction

A synthetic benchmark is a program that simulates the behavior of some typical programs but is not itself one of those programs. Many examples exist, including the Whetstone [Cur76] and Dhrystone [Wei84] benchmarks for scalar machines, Reed's communication benchmark for distributed-memory multiprocessors [Gru87], etc. The key attribute of these benchmarks is that they are programs that do not do any useful computing. They only exercise the machine in some "typical" way and report on its performance under such a workload.

In this paper we describe a method for constructing synthetic benchmarks for distributed memory multiprocessors. Each benchmark closely models the communication behavior of a real program. Each message has the same length and type and is sent from and to the same nodes. However, all computation unrelated to communication is replaced with delay loops that account for the time required to do the computation.

Synthetic benchmarks have many uses. A large collection of them can be used for quantifying the performance of existing, new, or experimental architectures. Porting the synthetic benchmarks described in this paper is particularly simple because all interprocessor communication uses PICL [Gei90], a portable, instrumented communication library. Thus, since all computation is replaced by delays, there is no porting effort required other than to adjust the delay times to reflect the computational speed of the new processor.

Since a synthetic benchmark is easy to construct, it can be used to experiment with alternate implementations or algorithms. In a sense, a simple prototype of the application can be constructed. For example, one can develop communication and computation structures, using simple delays to model the complex calculations and data structure manipulations. No input or output of data need be done, except for a few parameters such as problem size. Once the benchmark is run and its behavior and performance determined, modifications and improvements to the communication and computation structure can be made easily and simply.

A third application of synthetic benchmarks is to predict the behavior and performance of programs on new or hypothetical architectures. Since synthetic benchmarks are simple, porting them to new architectures can be done quickly, and the behavior and performance of the application determined with considerably less effort than porting the entire code. With hypothetical architectures, a simple simulation of the new architecture running the benchmark can predict the behavior and performance of both the machine and the application.

This paper reports on initial investigations into the construction and use of synthetic benchmarks done by the author while visiting at Oak Ridge National Laboratory.

2. Definitions

A synthetic benchmark is a program that models the communication and computation *structure* of a real program. It has the control structure of a real program, but does not do any actual computation. Instead each computation is replaced with an empty **for** loop that goes through enough iterations to consume the same amount of time necessary to do the computation it replaces.

All communication code remains, except that no real data is sent. Messages of the correct length and type are communicated, but the content of the messages is ignored. Computation that determines the length and type of messages remains in the code, as long as it is not data-dependent. (See below for a relaxation of this condition.)

No data input or output is done as the current focus of synthetic benchmarks is on the parallel, computational part. However a benchmark is usually parameterized by the problem size, which can be, for convenience, read as input. For example, programs doing matrix calculations would be parameterized by the size of the matrix. Other simple parameters that have an effect on the amount of computation or communication, other than actual data, can also be provided.

This approach works fine for programs that are not data-dependent. However, many programs are. In such cases, whenever the flow of the program is decided based upon a data dependent condition, the condition is replaced by a "well chosen" probability. For example, in an iterative algorithm that terminates when convergence occurs, the test for convergence can be replaced by a test whether a random variable is within an appropriate range (e.g., if the probability of convergence on a given iteration is 0.01, then the test would be whether a uniform random variable in the range 0 to 1 was less than 0.01). Another example might be the probability of the pivot element being on the diagonal in Gaussian elimination with partial pivoting.

3. Creating synthetic benchmarks

We describe three methods for creating synthetic benchmarks in this section. In all cases, the general method is to model the communication behavior exactly (subject to the constraints discussed below), and then to replace the computation taking place between explicit communications by delay.

In our examples, only the node program will be shown. A corresponding host program need only determine the problem size and machine size, start the node programs, send the problem size to the nodes, wait for completion, and terminate. No data need be read, distributed, collected and/or written. Consequently a very simple host program suffices.

Delay is added by inserting an empty **for** loop at the appropriate place. The number of iterations the loop must go through is determined in several ways. In all cases, however, it is necessary to know how many iterations are required to delay for a given length of time – say one millisecond. Since the resolution of the clock is one millisecond on the Intel iPSC/2, the multiprocessor used for our experiments, the loop was run for one million iterations in order to get an estimate for the time per iteration accurate to three decimal places. That many iterations took 0.761 seconds, resulting in 761 iterations per millisecond (0.761 iterations per microsecond). There was approximately a one microsecond startup time for the loop, but for simplicity and because delays are usually much longer than a microsecond, this overhead was ignored.

3.1. From scratch

One method for creating a synthetic benchmark program is to write one from scratch. This approach would be appropriate not only for creating a benchmark, but also for experimenting with trial implementations, or as a step in the development of an application.

The procedure is similar to writing a real application, except that none of the actual computing and input or output of data would have to be done, thus simplifying the task. In particular, complex data structures and/or intricate indexing are eliminated. All computing is simply replaced by delay loops whose delay time can be estimated from the complexity of the operations being replaced.

As an example, the synthetic benchmark below comes from the Fan-Out algorithm for solving triangular systems [Kuc76]. Pseudo code for the algorithm is given in Fig. 1.

```

for  $j := 1$  to  $n$  do
  if  $j \in \text{myrows}$  then
     $x_j := b_j / L_{jj};$ 
  endif;
  fan-out( $x_j, \text{map}(j)$ );
  for  $i := j + 1$  to  $n$  do
    if  $i \in \text{myrows}$  then
       $b_i := b_i - x_j L_{ij};$ 
    endif;
  end for;
end for

```

Figure 1: Pseudo code for the Fan-Out algorithm.

Rows of the matrix are distributed among the processors. $\text{map}(j)$ is a function that returns the processor containing row j . The result of the operation fan-out(t, root) is

```

# define COMPUTE(x) {int qi,qj; qj=x; for(qi=0; qi<qj; qi++);}
# define USEC *0.761
# define C (18.5 USEC)

int nodes, me, host, n;
int bufsize;
char buf[65536];

main() {
    int i, j;

    open0(&nodes, &me, &host);
    recv0(&n, sizeof(int), 999999);

    for (j=0; j<n; j++) {
        if (j%nodes == me)
            COMPUTE (C);
        fanout(j, j%nodes);
        for (i=j+1; i<n; i++)
            if (i%nodes == me)
                COMPUTE(2*C);
    }
    send0(buf, sizeof(int), 0, host);
    close0();
}

fanout(j, root)
int j, root;
{
    int i;

    if (root == me)
        send0(buf, sizeof(double), j, (me+1)%nodes);
    else {
        recv0(buf, sizeof(double), j);
        if ((me+1)%nodes != root)
            send0(buf, sizeof(double), j, (me+1)%nodes);
    }
}

```

Figure 2: Synthetic program for Fan-Out algorithm.

that processor *root* sends *t*, and all other processors receive *t*. Figure 2 contains the code for the synthetic benchmark.

The first three **define** statements implement the **COMPUTE** construct. The initial **open0** and **recv0** statements get the parameters for the problem, and the **send0** and **close0** inform the host that the node is done and terminate the node.

The fan-out function is implemented as a ring algorithm, although many other algorithms could have been chosen (e.g., recursive doubling). The root node sends the data to the next node in the ring. Each of the other nodes receives the data, and sends it to the next node in the ring, except for the node preceeding the root node in the ring, which simply receives the data.

Note that the algorithm is simple, and so is the code. In fact, compared to a "real" program for this algorithm, the synthetic code is much simpler, not having to deal with the input and output of data and results, calculations, and the sending and receiving of actual data.

3.2. From an existing program

Another approach is to create a benchmark from an existing program for which the source code is available. One would use this method for adding a program from a new type of application to a benchmark suite, or for constructing a simulation for predicting the performance of the program on a hypothetical machine.

Appendix A is a listing of an existing, complete program for solving a dense system of linear equations using Gaussian elimination with partial pivoting. Figure 3 contains a synthetic benchmark for the same program, where **COMPUTE**, **USEC**, and **C** are defined as in Fig. 2.

Note the following points:

- (1) The subroutine structure has been eliminated to produce a small example for this paper. Typically the original structure of the program would be preserved.
- (2) Pivoting is data-dependent. However the synthetic program is not (and cannot be) and therefore does not simulate precisely the behavior of a real program. Instead, whether to swap rows or not is contingent on the value returned from a random number generator (called **uniform**) that returns a real value in the range 0 to 1. In this example, the assumption was made that the probability that the pivot element is on the diagonal is 0.05, thereby causing the program to consume time associated with the swapping of rows 95 percent of the time. Another possibility would have been to choose the row in which the pivot was found randomly, but in this implementation it makes no difference to the behavior or performance of the program.


```

int nodes, me, host, i, j, k, n;
char buf[65536];

main() {
    open0(&n, &me, &host);          /* initialize */
    recv0(&n, sizeof(int), 0);

    for (i=0; i<n; i++) {
        if (i%nodes == me) {
            COMPUTE(2*C*(n-i));      /* find pivot */
            for (to=me+1; to<me+nodes; to++) {
                send0(buf, sizeof(int), i, to%nodes);
                send0(buf, n*sizeof(double), i, to%nodes);
            }
        }
        else {
            recv0(buf, sizeof(int), i);
            recv0(buf, n*sizeof(double), i);
        }
        if (uniform(0.0,1.0) > 0.05)
            for (k=i; k<n+1; k++)      /* swap rows */
                if (k%nodes == me)
                    COMPUTE(C);
        for (k=n; k>=i; k--)          /* elimination */
            if (k%nodes == me)
                COMPUTE(C*(n-i));
    }
    if (n%nodes == me) {              /* back substitution */
        for (j=n-1; j>=0; j--)
            if (j%nodes == me)
                COMPUTE(C*j);
            else
                if (j != 0) {
                    recv0(buf, n*sizeof(double), n+j);
                    COMPUTE(C*j);
                }
    }
    else
        for (j=me; j<n; j+=nodes)
            if (j != 0) {
                COMPUTE(C*j);
                send0(buf, n*sizeof(double), n+j, n%nodes);
            }

    send0(buf, sizeof(int), 0, host); /* terminate */
    close0();
}

```

Figure 3: Synthetic program for Gaussian elimination with partial pivoting.

- (3) All messages are sent and received from the same dummy buffer (except n , the size of the problem).
- (4) All computation is compressed into `COMPUTE` statements, using $C = 18.5$ microseconds as the average time for a single floating point operation, including memory access, indexing, etc. A more accurate function could have accounted for each operation separately.

3.3. From observed behavior

The third approach is to create a benchmark by modeling the observed behavior of an existing program. One might consider doing this if the source code for the application were not available (e.g., proprietary), or if the source code were extremely difficult to understand (e.g., poorly written). While no technique for constructing a program purely from observed behavior can be guaranteed to work, for matrix-oriented data-independent algorithms, the following steps can be followed:

First, model the communication behavior of the program:

- (1) Pick values of p (the number of processors) and n (the size of the problem) such that all of the following values are distinct: $p, n, n/p, \log_2 p$ (e.g., $p = 8, n = 32$).
- (2) Run the program for these p and n , get trace files, and produce graphs of the communication behavior. A visual representation of the communication helps tremendously, and examples of graphs produced for a particular application are shown in Appendix C.
- (3) Hypothesize the structure of the communication and write code that will produce it.
- (4) Verify that the code produces the correct behavior for the p and n chosen, and if so, also for other p and n satisfying the relationship above. If not, modify the code until it works for many combinations of p and n by repeating steps (2) and (3).

Once the communication behavior is modeled, the computation times can be simulated with delay loops. The following steps can be followed:

- (1) Get patterns of computation times between each send/recv for a few different n and p .
- (2) Hypothesize general structure of a function for compute time, usually a function of n, p , or both. Use some statistical method (e.g., least squares) to determine constants.

- (3) Verify for several n and p as done for communication structure.

We now show the procedure used to create a synthetic code for a particular, unknown algorithm. It turned out that the unknown algorithm is a triangular-system solver using a column-oriented wavefront vector-sum algorithm described in [Hea88] (the original code for this program is shown in Appendix B). However, neither the problem nor the algorithm nor the code were known at the time the synthetic code was being produced. Only an executable version of the program was available. Figure 5 contains the final synthetic code for this algorithm.

The table below shows the results of running the real program and the synthetic program for the various numbers of processors and sizes of problem.

		Problem Size		
		64	128	256
Nodes	4	160/162	605/622	2372/2439
	8	95/93	322/328	1206/1243
	16	72/68	190/191	645/660

Figure 4: Execution times in milliseconds for original program/synthetic program for wavefront algorithm.

Processor utilization curves for the real program and the synthetic program are displayed in Fig. 6 and Fig. 7, respectively. As can be seen, the curves do not differ significantly.

The procedure used for constructing this benchmark was as follows:

- (1) Ran the program for $p = 8$ and $n = 64$ and collected a communication trace.
- (2) Determined by examining the trace file that nodes were communicating in a ring topology with their neighbors in gray-code order (i.e., $0 \rightarrow 1 \rightarrow 3 \rightarrow 2 \rightarrow 6 \rightarrow 7 \rightarrow 5 \rightarrow 4 \rightarrow 0$). Verified this by running the program for $p = 4$ and $p = 16$.
- (3) Constructed message type and length graphs as shown in Appendix C. Each graph shows, for each node, the order in which messages were sent and received, and either length or type. Messages sent are indicated by lines starting at 0 and going positive; messages received are indicated by lines starting at 0 and going negative. The length of the line indicates the type (type graphs) or length (length graphs). The left to right order of the lines is the order in which messages

```

#define COMPUTE(x) {int qi,qj; qj=(x); for(qi=0; qi<qj; qi++);}
#define MSEC *761.267

int p, me, host, n, bufsize;
char buf[65536];

main() {
    int i, msgs, junk;
    int rnext, oldtype, imsgs, len, type, node;

    open0(&p, &me, &host);
    recv0(&n, sizeof(int), 999999);

    rnext = gray0((ginv0(me)+1)%p);
    msgs = (n*n+n*p)/(8*p) - (n/p)*(ginv0(me)/4);
    oldtype = 0;
    imsgs = 0;

    if (me == 0) {
        COMPUTE(0.01220 MSEC);
        send0(buf, 12, 1, rnext);
        for (i=1; i<n/4; i++) {
            COMPUTE(0.01220 MSEC);
            send0(buf, 16, 1, rnext);
        }
        imsgs = n/4;
    }

    for (i=imsgs; i<msgs; i++) {
        COMPUTE(0.13664 MSEC);
        recv0(buf, 16, ANY);
        recvinfo0(&len, &type, &node);
        if (type != oldtype) {
            len -= 4;
            oldtype = type;
        }
        if (len != 0) {
            COMPUTE(0.01220 MSEC);
            send0(buf, len, type+1, rnext);
        }
    }

    send0(&junk, sizeof(int), 0, host);
    close0();
}

```

Figure 5: Synthetic program for wavefront algorithm.

Figure 6: Processor utilization curve for the wavefront algorithm.

Figure 7: Processor utilization curve for synthetic model of wavefront algorithm.

were sent/received. The distance between lines is constant; it does not reflect the actual time between sends and/or receives. Graphs were drawn for each node for several values of p and n and were used in many of the following steps.

- (4) Ran for several values of p and n and found that the message lengths were always 4, 8, 12 or 16, and were independent of both p and n .
- (5) Noticed that node 0 started by sending $n/4$ messages. The first message was always length 12, the remaining always length 16. Following that, each node (including 0) received a message from its predecessor in the ring and then immediately sent a message to its successor.
- (6) The type of the message sent was always one more than the type of the message received.
- (7) If a node received a message of a new type, it would send a message that was 4 bytes shorter than the one it received, otherwise it would send a 16-byte message. If the message to be sent would have 0 length, it was not sent.
- (8) From this information, wrote an initial synthetic program to mimic this behavior. The initial program worked fine except for termination. To fix it, the total number of messages received by each node was determined to be

$$\frac{n^2 + (n/p)}{8p} - \frac{n}{p} \cdot \frac{\text{pos}(me)}{4},$$

except for node 0 which received $n/4$ fewer, where me is the node number of a given node and $\text{pos}(me)$ is the position in the gray-code ring (node 0 is at position 0).

- (9) The program was modified and run for several values of p and n , and the message traces compared with the real program (i.e., the order, types and lengths were compared for every node). Since they agreed in all cases, the program was assumed to have the correct communication structure.
- (10) Next, trace files were examined for the computational times between sends and receives. It appeared that the times between the send and receive and between the receive and send were both (different) constants, independent of both p and n .
- (11) From the trace files, the average delays were computed and inserted into the synthetic program. The program was run for several p and n and the overall run

times compared with the real program. After some fiddling with the times (they were too long initially), the final values shown in the program in Fig. 5 were determined.

The experience of constructing this and other synthetic benchmarks in this way pointed out the following things. First, the communication graphs were of immense help in finding patterns. It would have been difficult, if not impossible, to determine what was happening otherwise. Pictures are essential for discovering patterns.

Second, determining the delay times was a difficult task, mostly due to the low resolution of the clock on the iPSC/2 (1 millisecond). The eventual values for the delays were considerably less than a millisecond, which is much less than the clock resolution. To get the run times of the synthetic program to match those of the real program, much fiddling had to be done. To be more accurate, a clock with resolution on the order of a microsecond would be necessary.

Despite this difficulty, the synthetic benchmarks constructed did match real programs' run times to within five percent. As a further verification of the validity of this method, the programs were moved to an Ncube/3200. After a small adjustment of the computation delay times (almost in exact proportion to the relative computation rates of the iPSC/2 and the Ncube/3200), the synthetic programs had run times very close to the the real programs on the Ncube/3200 for a large range of p and n .

4. Performance prediction using synthetic benchmarks

Performance prediction of new and hypothetical architectures is another use for synthetic benchmarks. Constructing a benchmark suite, and using synthetic benchmarks for implementation prototyping are fairly straightforward and will not be discussed further.

Predicting the performance of an algorithm represented as a synthetic benchmark on a new architecture requires two things. First, the communication function calls must be translated into the proper form for the new architecture, unless the new architecture is compatible with the old, or unless a compatible library, such as PICL, was used in the synthetic benchmark. Second, the compute times must be determined for the new architecture. This may be as easy as simply scaling the delay constants by the ratio of the speed of an existing architecture, on which the benchmark already matches a real program, to the speed of the new architecture, measured by some simple unit such as megaflops. This will probably suffice for node processors that do not pipeline, do not have cache, etc., or as a first-order approximation for those that do. More accurate models of computation that take into account many aspects of the architecture of the node are necessary for more accurate compute times and, as a result, more

accurate performance prediction. We do not consider this here, except to say that as node architectures get more complex, more accurate models of compute times will be necessary. However, since the performance of distributed-memory machines is typically so much more dependent on communication performance than computation, the use of simple scaling should suffice for accurate predictions in most cases. The experience in running the benchmarks constructed for the iPSC/2 on the Ncube/3200 has shown that simple scaling works.

Predicting the performance of a hypothetical architecture requires the use of a simulator. The simulator can be simple if the architecture is simple or if high accuracy is not necessary. In particular, given the nature of the synthetic program, the simulator can be event driven, where an event is either the reception of a message or the termination of a `COMPUTE` statement. In this case the execution time of the simulation is proportional to the *number* of events. This often eliminates one of the common drawbacks of simulation, long execution times. Using an event driven simulator, a communication taking hundreds of microseconds on a real machine can be simulated in less time, and a computation taking milliseconds, seconds or even minutes or hours can be simulated in a few microseconds — the time it takes to evaluate the expression in the `COMPUTE` statement and post an event in the event queue of the simulator.

The fortunate result of fast performance predictions with simulators using synthetic benchmarks is that many simulations, with varying parameters, can be done in a reasonable time. Instead of just getting a performance prediction of a single program on one data set and one set of architectural parameters, many programs, with different problem-dependent parameters (like problem size), and varying architectural parameters (like communication or computation speed), can be simulated in a reasonable time. This produces more insight, not only into performance, but into relationships between program, problem parameters, and architectural characteristics.

Using synthetic benchmarks for performance prediction produces more than simply a measure of the execution time of the algorithm on the new or hypothetical architecture. It also produces the lower level, detailed behavior of the algorithm. This can be useful in determining heretofore unseen bottlenecks in the algorithm, processor utilization as a function of time, interconnect congestion, etc.

5. Summary

In this paper we have shown that simple programs can emulate the behavior of more complex programs to a high degree of accuracy. This means that constructing a suite of small, easily ported benchmark programs would be quite easy using at least two of the methods shown herein. A large suite, encompassing a wide variety of applications,

would be of immense utility in characterizing the performance of parallel architectures.

The simplicity of construction of synthetic codes also makes them ideal for quick prototyping of large, complex real applications. One can focus on the larger issues of data partitioning and computation and communication algorithm choice without getting bogged down in the details of implementing complex computations and data structures.

Finally, predicting the behavior and performance of an application on new or hypothetical architectures through the use of a synthetic benchmark of the application makes flexible and efficient performance prediction possible.

5.1. Further work

More work remains to further refine the techniques for creating synthetic benchmarks. In particular, the conversion of programs that are data-dependent into synthetic codes using statistical or probabilistic methods needs to be further investigated. Another interesting question is how one would incorporate caching, vectorization, and pipelining of the computational aspect of programs.

A major goal is to construct a large suite of synthetic benchmarks covering a wide variety of areas of study and many diverse algorithms. Work to date has focused on matrix algorithms, but algorithms for other numerical applications as well as non-numerical algorithms are necessary as well.

The follow-on to the construction of the benchmark suite is using it in the characterization of the performance of various distributed-memory parallel architectures. Because synthetic benchmarks are naturally parameterized for problem size, machine size, alternate communication implementations, and statistical properties, much would be gained by quantifying the impact of variation in these parameters on various architectures and algorithms.

Acknowledgments

The author thanks Mike Heath and Oak Ridge National Laboratory (ORNL) for providing the opportunity and resources for this work. Pat Worley provided the inspiration, much coaching, some criticism, but mostly needed encouragement (and converted the entire paper from troff to TeX). Conversations with Mike and Al Geist helped form and refine many of the ideas. Tom Dunigan's suggestions improved the readability of the paper.

References

- [Cur76] H. J. CURNOW AND B. A. WICHMAN, *A synthetic benchmark*, Computer Journal, 19 (1976).
- [Gei90] G. A. GEIST, M. T. HEATH, B. W. PEYTON, AND P. H. WORLEY, *PICL: a portable instrumented communication library, C reference manual*, Tech. Report ORNL/TM-11130, Oak Ridge National Laboratory, Oak Ridge, TN, July 1990.
- [Gru87] D. C. GRUNWALD AND D. A. REED, *Benchmarking hypercube hardware and software*, in Hypercube Multiprocessors 1987, M. T. Heath, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987, pp. 169-177.
- [Hea88] M. T. HEATH AND C. H. ROMINE, *Parallel solution of triangular systems on distributed-memory multiprocessors*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 558-587.
- [Kuc76] D. J. KUCK, *Parallel processing of ordinary programs*, in Advances in Computers, 15 (1976), pp. 119-179.
- [Wei84] R. WEIKER, *Dhrystone: A synthetic systems programming benchmark*, Comm. ACM, 27 (1984), pp. 1013-1030.

A. Source code for Gaussian elimination with pivoting

```

/*-----
Solve dense set of simultaneous equations using
Gaussian elimination with partial pivoting
-----*/

# define N 512                                /* maximum number of equations */

# define NTYPE 1                              /* message types */
# define COLTONODE 2
# define COLTOHOST 3
# define PIVOT 4
# define SUBST 5

double *a[N+1];                               /* variables */
double col[N];
int n = 0;
int me,nodes,host;
int bufsize;

# define COL(x) ((x-me)/nodes)
# define A(x,y) (a[y][x])

/*-----

main driver - get misc. info and data, solve, return results
-----*/

main() {

    int i;

    /* get started */

    openOP(&nodes, &me, &host);
    recvO(&nodes, sizeof(int), NTYPE);
    recvO(&n, sizeof(int), NTYPE);
    bufsize = n * sizeof (double);

    /* get the data */

    recvm();

    /* solve the equations */

    solve();

```

```

/* send back results */

sendm();
close0();
}

/*-----
receive the columns of the matrix for this node
-----*/

recv0() {
    int j, cj;

    /* get the columns for this node */

    for (j=me; j<n+1; j+=nodes) {
        cj = COL(j);
        a[cj] = (double *) malloc(bufsize);
        recv0(a[cj], bufsize, COLTONODE);
    }
}

/*-----*/

driver loop for solving equations

/*-----*/

solve() {
    int i,j,k;

    /* for each column of the matrix (except the last) */

    for (i=0; i<n; i++) {
        /* either ... */

        if (i%nodes == me) {
            /* find the pivot and send it and column to others */

            j = findmax(i);
            sendcol(i,j);
        }
        else {
            /* or receive the pivot and column from another */

```

```

        j = recvcol(i);

        /* exchange rows */
        swaprow(i,j);

        /* do eliminations */
        elim(i);
    }

    /* do back substitutions */

    subst();
}

/*-----*/

    find pivot row

-----*/

findmax(i)
int i;
{
    int ci,j,k;
    double max;

    tracemark(0);
    ci = COL(i);
    max = -1.0;
    j = -1;
    for (k=i; k<n; k++)
        if (fabs(A(k,ci)) > max) {
            max = fabs(A(k,ci));
            j = k;
        }
    if (max == 0.0)
        printf("ERROR: Singular Matrix");
    tracemark(1);
    return(j);
}

/*-----*/

    send index of pivot row, and pivot column to other nodes

-----*/

```

```

sendcol(i,p)
int i,p;
{
    int ci,j,to;

    /* make a copy of the pivot column */

    ci = COL(i);
    for (j=i; j<n; j++)
        col[j] = A(j,ci);

    /* for each other node ... */

    for (to=me+1; to<me+nodes; to++) {

        /* send index of pivot row */

        send0(&p, sizeof(int), PIVOT+i, to%nodes);

        /* send pivot column */

        send0(col, bufsize, PIVOT+i, to%nodes);
    }

}

/*-----
   receive index of pivot row and pivot column from other node
   -----*/

recvcol(i)
int i;
{
    int j,k;

    /* get index of pivot row */

    recv0(&k, sizeof(int), PIVOT+i);

    /* get pivot column */

    recv0(col,bufsize,PIVOT+i);

    return(k);
}

```

```

/*-----
    swap rows in columns in this node
-----*/

swaprow(i,j)
int i,j;
{
    int ck,k;
    double temp;

    /* don't swap if unnecessary */

    if (i == j)
        return;

    /* swap in pivot column */

    tracemark(10);
    temp = col[i];
    col[i] = col[j];
    col[j] = temp;

    /* swap in all other columns */

    for (k=i; k<n+1; k++) {
        if (k%nodes != me)
            continue;
        ck = COL(k);
        temp = A(i,ck);
        A(i,ck) = A(j,ck);
        A(j,ck) = temp;
    }
    tracemark(11);
}

/*-----

    do elimination step
-----*/

elim(i)
int i;
{
    int ck, j, k;
    double coli, temp;
    double * acol;

```

```

/* reduce */

tracemark(20);
coli = col[i];
for (k=n; k>=1; k--) {
    if (k%nodes != me)
        continue;
    ck = COL(k);
    acol = a[ck];
    temp = acol[i]/coli;
    for (j=i+1; j<n; j++)
        acol[j] -= col[j]*temp;
}

/* normalize pivot row */

for (k=n; k>=1; k--) {
    if (k%nodes != me)
        continue;
    A(i,COL(k)) /= coli;
}
tracemark(21);
}

/*-----
do back substitution
-----*/

subst() {
    int cj,cn,i,j,k;

    /* if this node contains the last column */

    cn = COL(n);
    if (n%nodes == me) {

        /* do substitutions from each column */

        for (j=n-1; j>=0; j--) {

            /* if column in the same node */

            cj = COL(j);
            if (j%nodes == me) {

                /* just do substitutions */

```



```

        tracemark(30);
        for (i=0; i<j; i++)
            A(i,cn) -= A(i,cj)*A(j,cn);
        tracemark(31);
    }
else

    /* get column from another node
    and do substitutions */

    if (j != 0) {
        recv0(col, bufsize, SUBST+j+n);
        tracemark(30);
        for (i=0; i<j; i++)
            A(i,cn) -= col[i]*A(j,cn);
        tracemark(31);
    }
}
else {

    /* send columns to the node doing the substitutions */

    for (j=me; j<n; j+=nodes)
        if (j != 0) {
            cj = COL(j);
            for (i=0; i<j; i++)
                col[i] = A(i,cj);
            send0(col, j*sizeof(double), SUBST+j+n, n/nodes);
        }
    }

    /*-----

    send results to host

    -----*/

    sendm() {

        /* if this node does not contain last column, return */

        if (n/nodes != me)
            return;

        /* otherwise send the last column */

        send0(a[COL(n)], bufsize, COLTOHOST, host);
    }

```

B. Source code for wavefront algorithm

```

/*-----
Solve triangular system of equations using column oriented wavefront
vector-sum algorithm
-----*/

col_wav_0 ( n, ncols, map, mycols, col, b, segment )
{
    int n, ncols, *map, *mycols, segment ;
    float **col, *b ;

    int i, j, k, m, me, lim, tag, bytes, cnt, node, type ;
    float *z ;

    segment = (segment > n) ? n : segment ;
    bytes = segment*sizeof(float) ;
    z = (float *)malloc0(bytes) ;

    j = 0 ;
    for ( k = 0 ; k < n ; k++ ) if (map[k] == me) {
        tag = 1 ;
        m = 0 ;
        while (m < n-k) {
            if (k > 0) {
                recv0(z, bytes, k) ;
                recvinfo0(&cnt, &type, &node) ;
            }
            else {
                cnt = segment*sizeof(float) ;
                for ( i = 0 ; i < segment ; i++ ) z[i] = 0 ;
            }
            lim = cnt/sizeof(float) ;
            if (lim > n-m) lim = n-m ;
            if (tag) {
                b[j] -= z[0] ;
                b[j] /= *col[j] ;
                m++ ;
            }
            for ( i = tag ; i < lim ; i++ , m++ )
                z[i] += b[j] * *(col[j]+m) ;
            if (k < n-1 && lim-tag > 0)
                send0(&z[tag], (lim-tag)*sizeof(float),
                    k+1, map[k+1]) ;
            tag = 0 ;
        }
        j++ ;
    }
    free(z) ;
}

```

C. Graphs of message types and lengths for wavefront algorithm

Figure 8: Graph of message types for node 0.

Figure 9: Graph of message types for node 1.

Figure 10: Graph of message types for node 3.

Figure 11: Graph of message types for node 2.

Figure 12: Graph of message types for node 6.

Figure 13: Graph of message types for node 7.

Figure 14: Graph of message types for node 5.

Figure 15: Graph of message types for node 4.

Figure 16: Graph of message lengths for node 0.

Figure 17: Graph of message lengths for node 1.

Figure 18: Graph of message lengths for node 3.

Figure 19: Graph of message lengths for node 2.

Figure 20: Graph of message lengths for node 6.

Figure 21: Graph of message lengths for node 7.

Figure 22: Graph of message lengths for node 5.

Figure 23: Graph of message lengths for node 4.

- END -

DATE FILMED

11 / 16 / 90