CS 442 Introduction to Parallel Processing
Project 3: Wave Equation

Matt Bohnsack and Erik Barry Erhardt

May 5, 2006

Contents

1 Executive Summary **3**

2 Problem Description **4**

3 Programming Component **4**

4 Data Generation **5**
- 4.1 Governing equations 5
- 4.2 Initial conditions 5
- 4.3 Experimental region for scalability study 6
- 4.4 Hardware and systems software for scalability study 7

5 Results **8**
- 5.1 Visualization of solution to wave equation 8
- 5.2 Scalability 11

6 Conclusions **16**

A Appendix **17**

List of Figures

1 Photograph of Linux cluster (Machine B) 2
2 2D Wave Propagation vs Time 9
3 3D Wave Propagation vs Time 10
4 Machine A: Parallel Execution Time to 16 Processors 12
5 Machine A: Parallel Speedup and Efficiency to 16 Processors . 13
6 Machine B: Parallel Execution Time to 144 Processors 14
7 Machine B: Parallel Speedup and Efficiency to 144 Processors 15

List of Tables

1 Experimental region explored. 6

Figure 1: Photograph of Linux cluster (Machine B)

1 Executive Summary

For this project, we consider a parallel algorithm to implement a finite difference scheme solving a second-order wave equation in two dimensions with message passing of ghost cells and the Message Passing Interface (MPI). The parallel algorithm is implemented in `C` and MPI and run on a large number of processors on two similar parallel Linux clusters. Using this code, many independent parallel runs are conducted, and program speedup is explored.

The rest of this paper describes the problem and our implementation in greater detail, gives an overview of the parallel machines we ran on, discusses our experiment, presents plots of an example wave propagation and plots of speedup, and makes observations about the performance. Finally, conclusions are made and our code is presented in the appendix.

2 Problem Description

Many scientific applications rely on finite difference schemes over a mesh to approximate the solution to a dynamical system. This project implements the second-order wave equation in two dimensions over a square regular mesh. Parallelism is achieved by partitioning the square mesh into equal-sized rectangles, with each rectangle being assigned to a different process. Time progression is achieved through the solution of the wave equation using a finite difference scheme on a common five-point stencil (like a + sign); each point's new value is updated through its current value and the values of its four adjacent neighbors [Strikwerda] (p. 202). In order to update points on the border of neighboring rectangles, where the stencil would overhang, a single row of points is added to the borders of each rectangle, called the "ghost cells" [Quinn] (p. 326). At the beginning of each time step, neighboring processes need to send their border information into the appropriate ghost cells of the neighboring process. After the update, the time progression can occur. The stencil will obtain the correct values for the border points.

Initial conditions simulate a "tossing rocks in a pond" scenario, where the points of the mesh will be initialized to 0 except for single points which have a force applied to it. Boundaries are reflecting.

This is an interesting problem because (1) it simulates an easy to understand real-world phenomenon that can be visually verified and (2) it has a numerically intense algorithm that can greatly benefit from parallelization.

3 Programming Component

We first implement the serial algorithm for the finite difference wave equation from [Strikwerda] (p. 202). We extend this to a parallel algorithm by partitioning our mesh into N_x horizontal components and N_y vertical components to run as $N_p = N_x N_y$ equal-sized processes. Neighboring processes share their boundary information using a ghost cell exchange as required by the stencil [Quinn] (p. 326).

4 Data Generation

4.1 Governing equations

The second-order two-dimensional wave equation [Strikwerda] (p. 202 eq. 8.4.1) is given by

$$u_{tt} = a^2(u_{xx} + u_{yy})$$

where u_{tt} is the second derivative with respect to time, a^2 relates to the speed of propagation through the medium, and u_{xx} and u_{yy} is the second derivative with respect to the x and y spacial dimensions.

The simplest scheme for this equation [Strikwerda] (p. 202 eq. 8.4.2) is

$$\delta_t^2 v_{l,m}^n = a^2(\delta_x^2 v_{l,m}^n + \delta_y^2 v_{l,m}^n).$$

Expanding via the δ operator to the finite difference solution, and solving for the next time step gives

$$\begin{aligned} v_{l,m}^{n+1} &= a^2 \left(\left(v_{l-1,m}^n - 2v_{l,m}^n + v_{l+1,m}^n \right) \frac{\Delta t^2}{\Delta x^2} + \left(v_{l,m-1}^n - 2v_{l,m}^n + v_{l,m+1}^n \right) \frac{\Delta t^2}{\Delta y^2} \right) \\ &\quad + 2v_{l,m}^n - v_{l,m}^{n-1}, \end{aligned}$$

where Δt is the time step, and Δx and Δy are the mesh resolutions in the x and y directions.

The wave equation is second-order, which means that it relies on information from the previous two time steps to solve for the next time step. Therefore, on the first iteration, it is necessary to obtain a first-order approximation for the next time step. While there are many ways of doing this, we use a two-dimensional forward-time central-space scheme [Strikwerda] (p. 17 eq. 1.3.3)

$$v_{l,m}^{n+1} = a \left(\left(v_{l-1,m}^n - v_{l+1,m}^n \right) \frac{\Delta t}{\Delta x} + \left(v_{l,m-1}^n - v_{l,m+1}^n \right) \frac{\Delta t}{\Delta y} \right) + v_{l,m}^n.$$

4.2 Initial conditions

The mesh is initialized to 0 everywhere except a few points which are given values. Some points are given positive values, others negative, and of different magnitudes; some are near the center, and others near the boundary. The boundary condition is reflective, since it is interesting to see the waves reflect back onto themselves.

4.3 Experimental region for scalability study

Table 1 on page 6 summarizes the design space we explored. The grid size g is the resolution of the grid in one dimension, so the total problem size is g^2 on a unit square with uniform grid resolution over both dimensions. We used all combinations of $g \times N_p$.

Grid size g	x procs N_x	$\times$	y procs N_y	$=$	Total procs N_p
60	1	$\times$	1	$=$	1
120	2	$\times$	2	$=$	4
180	2	$\times$	3	$=$	6
240	3	$\times$	3	$=$	9
300	3	$\times$	4	$=$	12
360	4	$\times$	4	$=$	16
420	4	$\times$	5	$=$	20
480	5	$\times$	5	$=$	25
540	5	$\times$	6	$=$	30
600	6	$\times$	6	$=$	36
660	5	$\times$	8	$=$	40
720	5	$\times$	10	$=$	50
780	6	$\times$	10	$=$	60
	10	$\times$	10	$=$	100
	10	$\times$	12	$=$	120
	12	$\times$	12	$=$	144

Table 1: Experimental region explored.

4.4 Hardware and systems software for scalability study

Our data were generated on two separate, but similar Linux clusters.

The first machine (Machine A) was comprised of 32 computes nodes and a 2 Gbit/s Myrinet 2000 network. Each of this machine's compute nodes had two 1.2 GHz Pentium III CPUs and 1 GiB of RAM. We were able to explore problems up to 16 CPUs on this machine.

The second machine (Machine B) was comprised of 256 compute nodes and a 2 Gbit/s Myrinet 2000 network, each node having two 3.06 GHz Intel Xeon CPUs and 2 GiB of RAM. A photograph of this machine is shown in Figure 1 on page 2. We explored the full space from Table 1 on this machine, all the way up to 144 processors.

Both clusters ran the same software stack. The OS and kernel were RedHat Enterprise Linux release 4 and 2.6.9-22.0.2.ELsmp, and GM version 2.1.24 was used for the Myrinet driver. Our code was built with gcc version 3.4.4 and used MPICH version 1.2.6..14b-gm-2.1.24 for its MPI implementation. Jobs were submitted through a PBS scheduler and executed with a standard `mpirun` command.

5 Results

After verifying that our serial and parallel codes worked correctly, we experimented with different boundary conditions, such as absorbing and reflecting, and changed the position(s) of our initial point(s), until we identified a few interesting simulations for larger runs. Our code outputted data that allowed us to create contour plots of the solution and visualize the waves propagating through our grid at various time points.

Next we conducted two scalability studies. The first one was conducted on 1 to 16 processors, and the second one varied from 1 to 144 processors. These studies produced results that matched theory, achieving high degrees of scalability. We used data from the studies to create speedup and efficiency plots that demonstrate the benefits our finite difference algorithm receives from parallelization.

5.1 Visualization of solution to wave equation

The plots in Figure 2 on page 9 and in Figure 3 on page 10 show the propagation of the waves through the mesh at times $t = 0$ to $t = 0.5$ seconds at intervals of 0.05 seconds (solved with $\Delta t = 10^{-6}$). This mesh has total height and width of 1 unit, with 240 grid points in each direction ($\Delta x = \Delta y = \frac{1}{240}$), subdivided into 16 subsquares each run as a different process.

The initial conditions start with five points, which can be seen as dots on the first plot. Count the ranks of the processes from 0 to 15 as you would read, that is ranks 0 to 3 on the top row, 4 to 7 on the second row, etc. Rank 0 has two small points, one of magnitude 0.1 at grid point $(10, 10)$ and a second at -0.2 at $(60, 30)$. Rank 6 has 0.5 at its local grid point $(30, 30)$. Rank 8 has 1.0 at its local grid point $(30, 30)$. Rank 14 has -1.0 at its local grid point $(30, 30)$.

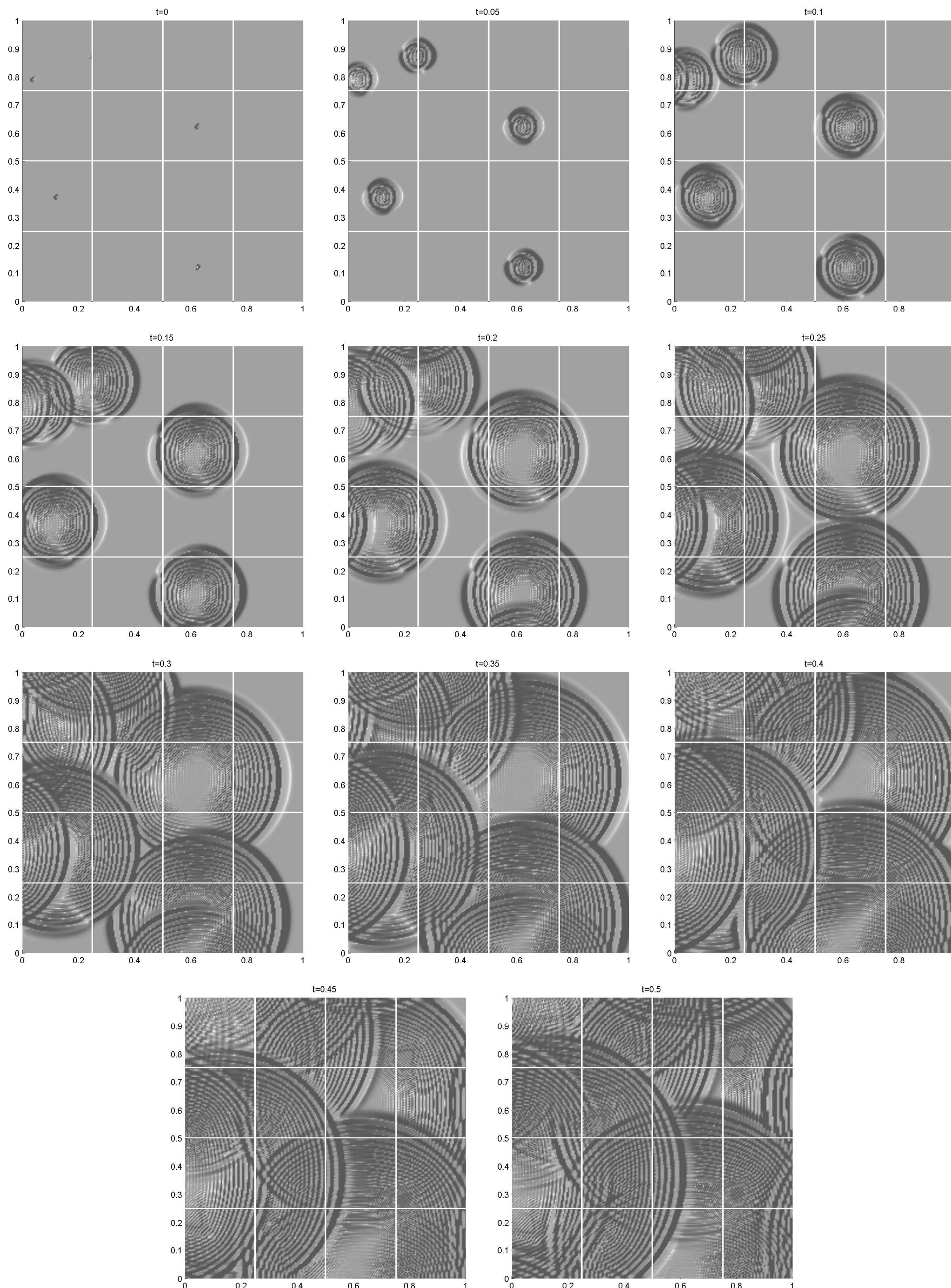

Figure 2: 2D Wave Propagation vs Time

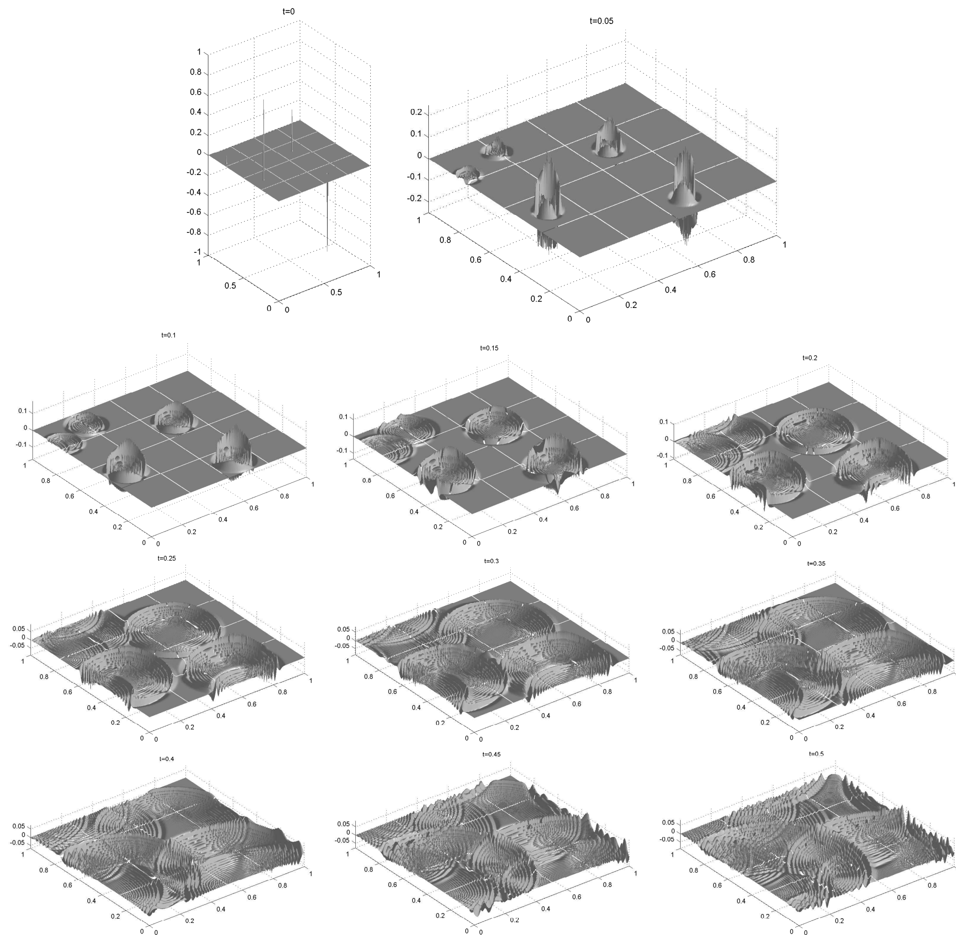

Figure 3: 3D Wave Propagation vs Time

5.2 Scalability

The plots in Figure 4 on page 12 show program run time of `wave.c` for various values of g over a range of processors (1 to 16) on Machine A. Increasing values of g create a finer mesh and a larger problem size. Both plots have number of processors on the horizontal axis and execution time on the vertical axis, but the top graph has time on a linear scale, while the bottom one shows time on a log scale.

In general, Figure 4 shows longer execution times for larger values of g and a decreasing execution times for any given g, as more processors are added to the problem.

The top plot in Figure 5 on page 13 shows parallel speedup relative to an equivalent run on a single processor of Machine A for 1 to 16 processors. The vertical axis represents the speedup (T_s/T_p), the horizontal axis shows number of processors, and the straight dot-dashed line represents a "perfect" linear speedup. The bottom plot in Figure 5 shows parallel efficiency ($T_s/(pT_p)$) on the vertical axis and number of processors on the horizontal axis for the same machine over the same range of processors.

Figure 5 shows excellent speedups (even superlinear in some cases over part of the range) and efficiencies in the 85 to 105% range for all granularities, except for $g = 60$, which is 15% less efficient than its nearest neighbor, at a 16 processor efficiency of $\approx 70\%$. The difference between $g = 60$ and the rest of the trials is a clear demonstration of scaled speedup, though the speedup differences between the other job granularity values don't show this as conclusively.

Figures 6 and 7 on pages 6 and 7 show information very similar to the Machine A plots, but they do so for Machine B from 1 to 144 processors.

In this larger scale Machine B experiment, $g = 60$ doesn't see any speedup past 60 processors. Other values of g show significant speedup all the way out to 144 processors, but fall short of ideal scaling past the 16 processors tested on Machine A. We expected to see scalability increasing consistently with g, as larger meshes translated into a higher ratios of computation to communication, but our data don't demonstrate this very clearly. Figure 7 has $g = 60$ at the bottom of the pack, $g = 120$ as the most scalable for the range between 60 and 120 processors, and $g = 180$ to 780 falling somewhere in between. As we were only able to run one trial of each data point, we suspect that various system variability issues added noise to our data that would have otherwise demonstrated a general scalability increase with increased values of g.

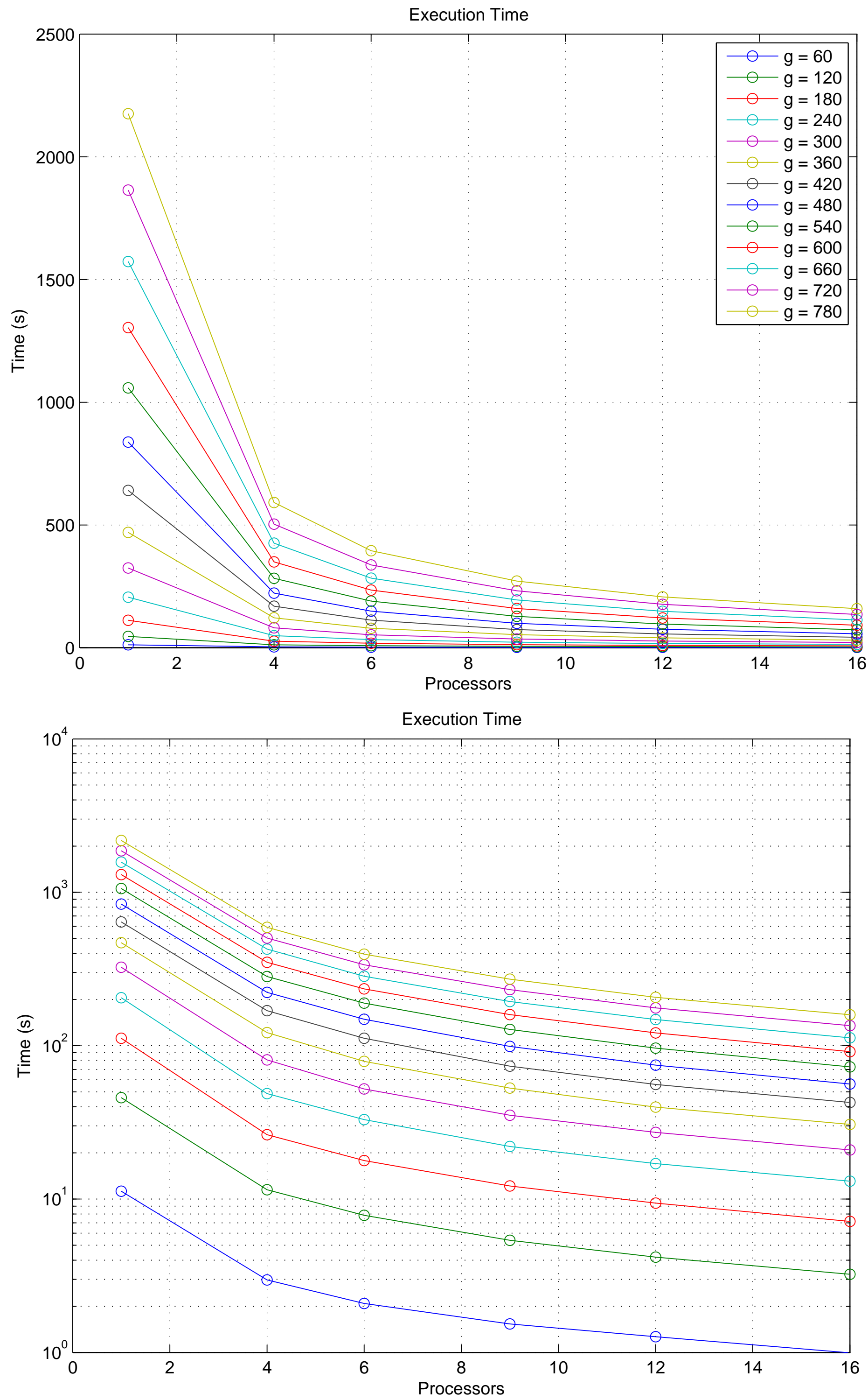

Figure 4: Machine A: Parallel Execution Time to 16 Processors

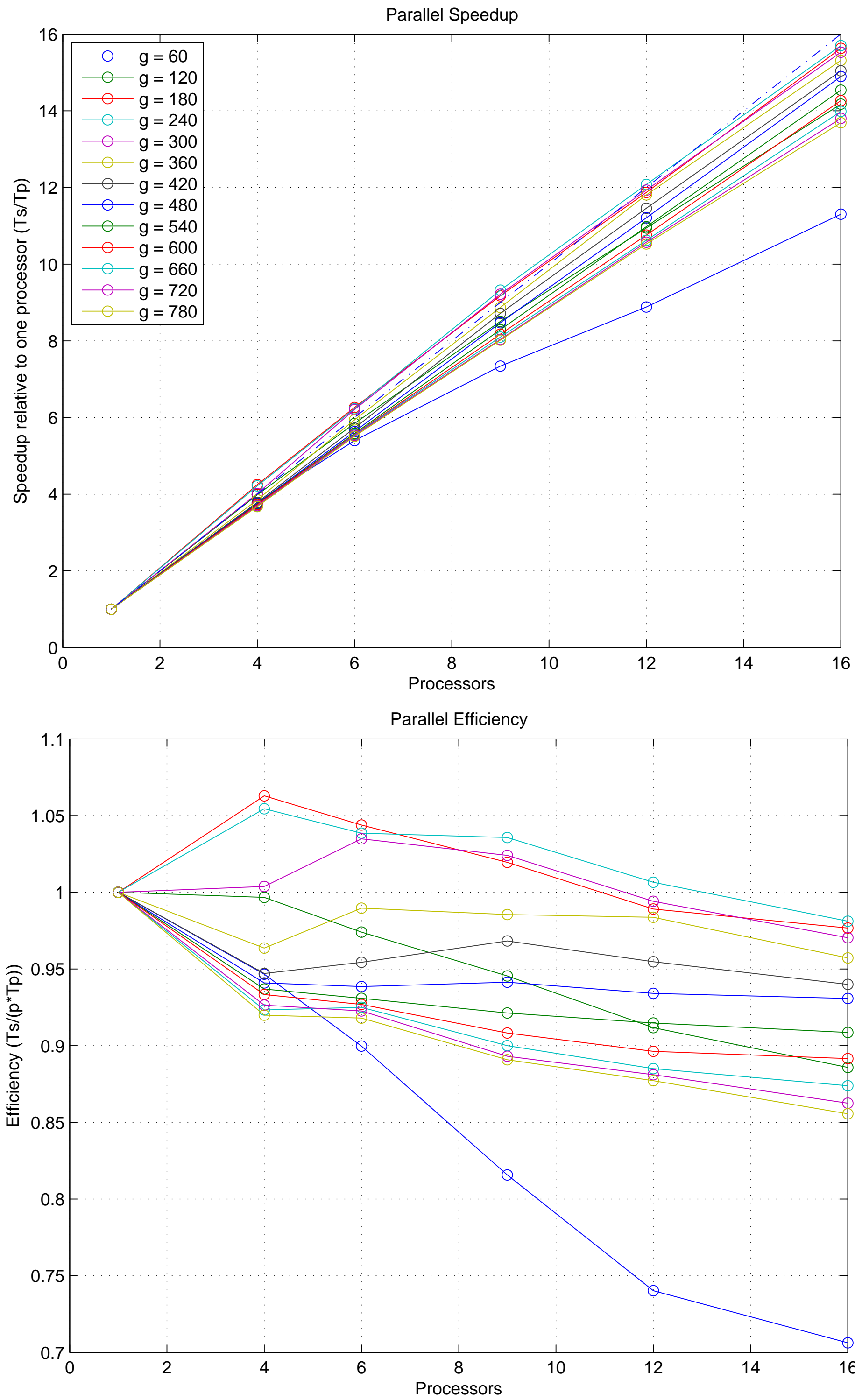

Figure 5: Machine A: Parallel Speedup and Efficiency to 16 Processors

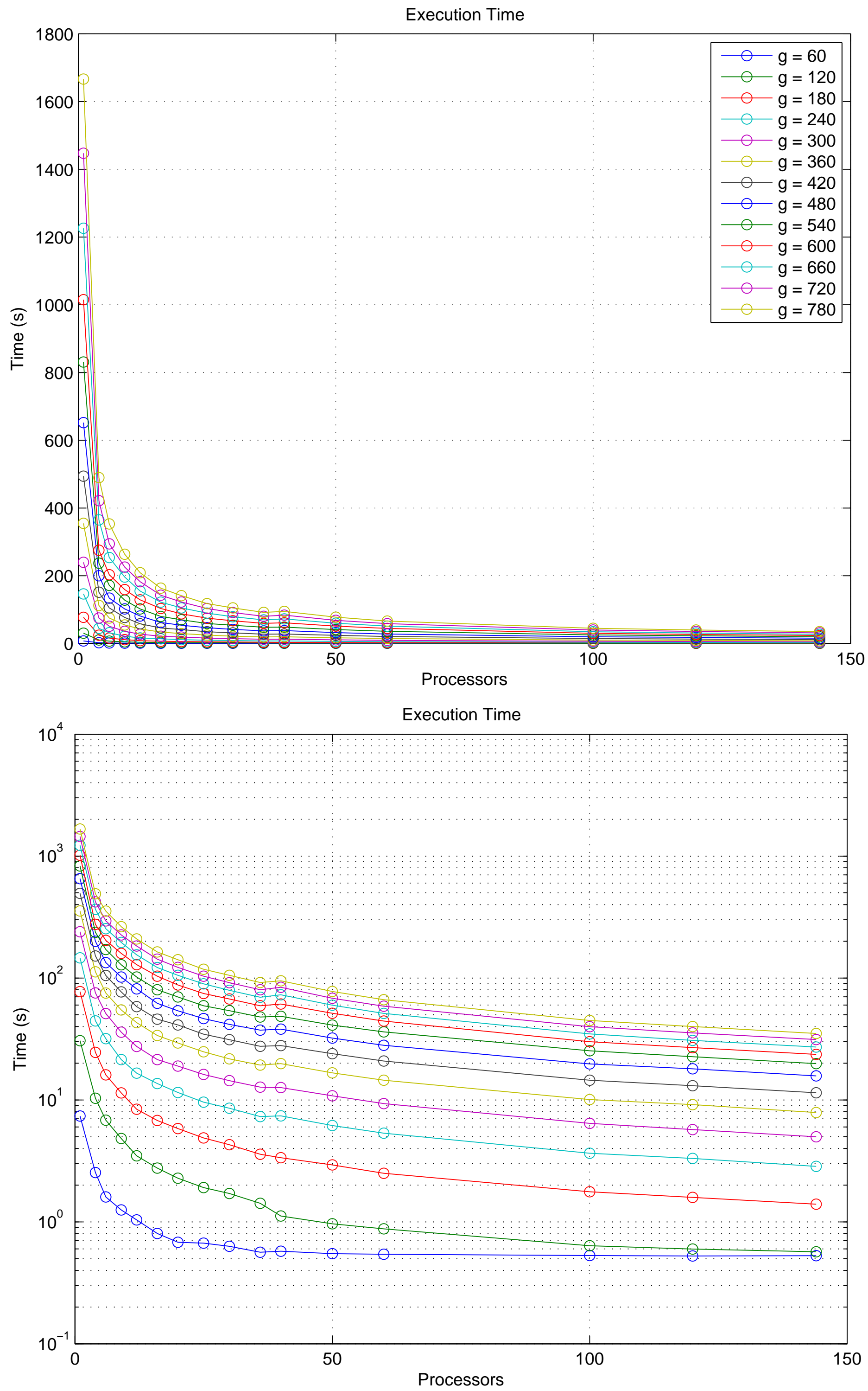

Figure 6: Machine B: Parallel Execution Time to 144 Processors

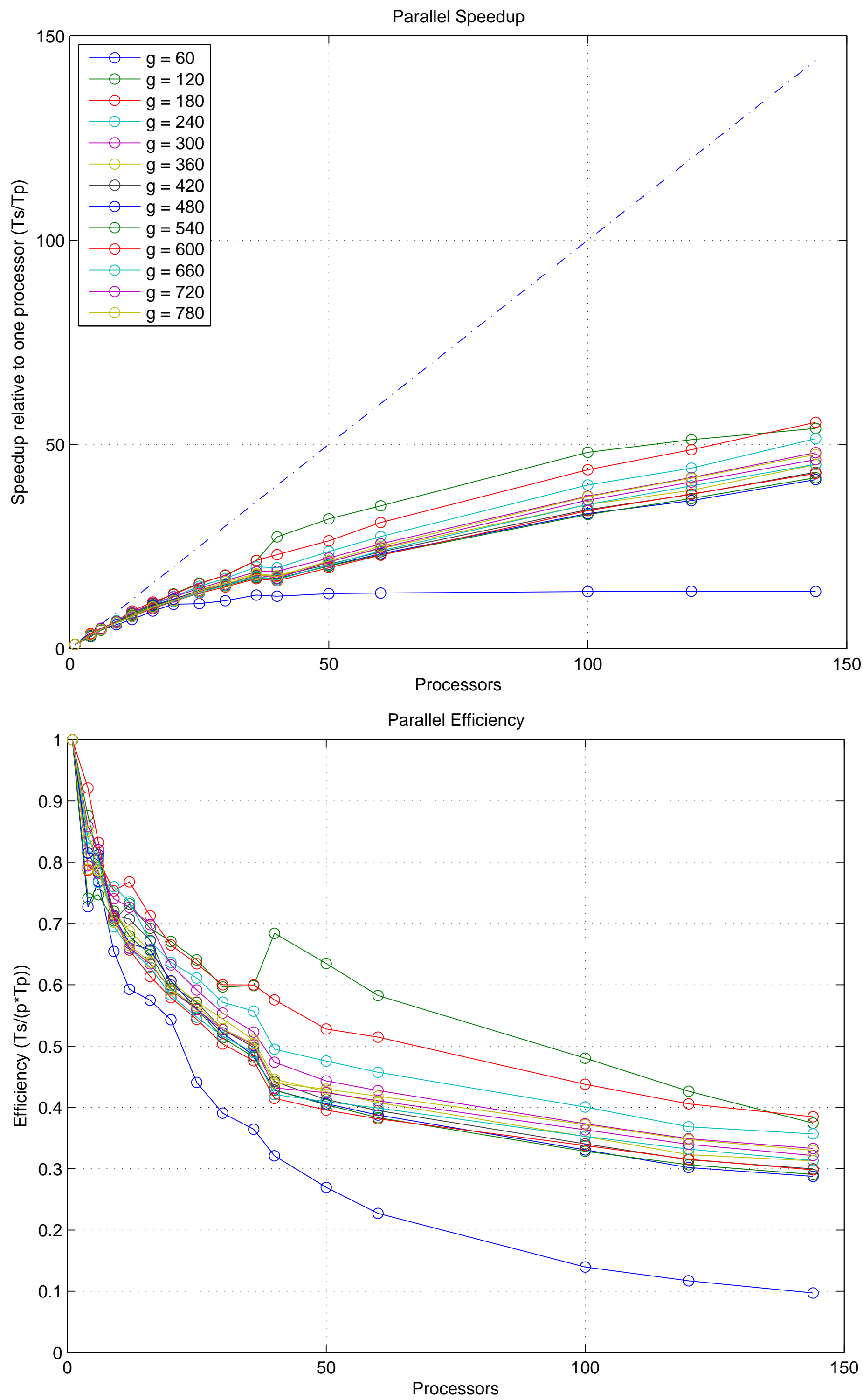

Figure 7: Machine B: Parallel Speedup and Efficiency to 144 Processors

6 Conclusions

In this project, we successfully used MPI to implement a parallel version of the two dimensional wave equation with ghost cell exchange. In doing so, we came to understand a good deal about message passing with MPI and how it can be used on production parallel systems. We were pleased that speedup was observed for all problem sizes.

Our data show that speedup from additional processors only begins to plateau for the smallest problem size, and the larger problem sizes demonstrate additional speedups are likely on larger number of nodes. This result is an exciting demonstration of scaled speedup and an excellent validation of Gustafson's Law.

References

[Strikwerda] John C. Strikwerda. Finite Difference Schemes And Partial Differential Equations. SIAM, 2nd ed.

[Quinn] Michael J. Quinn. Parallel Programming in C with MPI and OpenMP. McGraw-Hill, 1st ed.

A Appendix

Listing 1: Parallel implementation: `wave.c`.

```
/*****************************************************************************
Erik Barry Erhardt and Matthew Bohnsack
CS 442, Project 3, 4/30/2006
wave.c

   qsub -I -l nodes=4,walltime=3600
   touch wave.c
   make
   mpirun -np 16 -machinefile $PBS_NODEFILE ./wave 2 240 4 4

*****************************************************************************/

#include <mpi.h>
#include <math.h>
#include <stdio.h>

#define MAX_DIM 800

int rank, rank_x, rank_y, rank_down, rank_up, rank_left, rank_right;
int nprocs, nprocsx, nprocsy;
int nxg, nyg, nx, ny, n;
double length_x, length_y, delta_x, delta_y;
double PI, h, t0, tfinal, t, time_per_step;
int junk, error_flag=0, sw_method;
// this is our data matrix
double U[MAX_DIM+2][MAX_DIM+2], Ue[MAX_DIM+2][MAX_DIM+2], lu[MAX_DIM+2][MAX_DIM+2];
double U1[MAX_DIM+2][MAX_DIM+2];  // this is our data at time n-1

/*****************************************************************************/
int main(int argc, char *argv[]){
   int i, j, it;
   double time1, time2, timetotal, timesofar, timetogo;          // time the run

   MPI_Init(&argc,&argv);                    // initialize MPI
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);     // my rank number
   MPI_Comm_size(MPI_COMM_WORLD, &nprocs);   // number of processes

   time1 = MPI_Wtime();                      // first time t1

   junk = init(argc, &*argv);  /*******/                // initialize values

   if(sw_method==2){
      for (it=0; it<=n+1; it++){
         if(it % ((n-1)/10) == 0)
         { // output grid to a file
            double x_out[nx];
            double y_out[ny];
            double u_out[nx][ny];

            for (i=0; i<nx; ++i)   x_out[i]=( i + (rank_x*nx))/((double)(nxg-1));
            for (j=0; j<ny; ++j)   y_out[j]=( j + ((nprocsy-rank_y-1)*ny))/((double)(nyg-1));

            for (i=0; i<nx; i++) {
               for (j=0; j<ny; j++) {
                  u_out[i][j]=U[i+1][j+1];
               }
            }
            junk = output(nx,ny,x_out,y_out,u_out,t);
         }
         if((rank==0) && (it%10000==0)) {
            time2 = MPI_Wtime();                          // second time t2
            timesofar = time2 - time1;                    // total time is difference
            timetogo = (timesofar/it)*(n-it);
            printf("it=%d/%d, t=%f (%2.1f\%) (realtime so far=%f, to go=%f)\n", it, n, t,
                   (double)100*it/n, timesofar, timetogo);
         }
         junk = euler();
         t=t+h;
      }
   }

   time2 = MPI_Wtime();                                // second time t2
   timetotal = time2 - time1;                          // total time is difference
   time_per_step = timetotal/(n+2);
   if (rank==0) printf("time=%f, time_per_step=%f\n", timetotal, time_per_step);
   //if (rank==0) printf("time=%f\n", timetotal);

   MPI_Finalize();
}


/*****************************************************************************/
```

```
83  /* init *****************************************/
84  int init(int argc, char **argv){
85     int i, j;
86     int argc_counter, string_counter;
87     int result;
88     char argv1[81];
89
90     if(argc<5){
91         if(rank==0) printf("argc %d\n", argc);
92         if(rank==0) printf("Command line args: sw_method(1,2) nxg nprocsx nprocsy\n");
93         if(rank==0) printf("EXITING\n");
94         exit(0);
95     }
96
97     // 2 is part 2 for propagation of wave.
98     strcpy(argv1,argv[1]); sw_method = string2int(argv1);
99     strcpy(argv1,argv[2]); nxg       = string2int(argv1);
100    strcpy(argv1,argv[3]); nprocsx   = string2int(argv1);
101    strcpy(argv1,argv[4]); nprocsy   = string2int(argv1);
102
103    if(rank==0) {
104        printf("Params: sw_method=%d, nxg=%d, nprocsx=%d, nprocsy=%d\n", sw_method, nxg,
105               nprocsx, nprocsy);
106    }
107
108    PI=4*atan( (double)1 );                 // known value of pi
109    length_x=1; length_y=length_x;
110    nyg=nxg;
111    delta_x=length_x/(nxg+1); delta_y=delta_x;
112
113    if(nxg%nprocsx==0){
114        nx=nxg/nprocsx;
115    } else {
116        if(rank==0) {
117            printf("ERROR: nxg %% nprocsx!=0 (%d %d)\n", nxg, nprocsx);
118        }
119        error_flag++;
120    }
121    if(nyg%nprocsy==0) {
122        ny=nyg/nprocsy;
123    } else {
124        if(rank==0) {
125            printf("ERROR: nyg %% nprocsy!=0 (%d %d)\n", nyg, nprocsy);
126        }
127        error_flag++;
128    }
129
130    if(nprocs != nprocsx*nprocsy) {
131        if(rank==0) {
132            printf("ERROR: nprocs != nprocsx*nprocsy (%d != %d * %d), EXIT\n", nprocs, nprocsx,
133                   nprocsy);
134        }
135        error_flag++;
136    }
137
138    if(sw_method == 1){
139       if(rank==0) printf("Method 1, testing Laplacian.\n");
140       n=1;
141    };
142
143    if(sw_method==2 | sw_method==3){
144       if(rank==0) printf("Method 2, Wave.\n");
145       h=0.000001; t0=0.0; tfinal=0.5; t=t0;
146       // h=(pow(delta_x,2)/8)/4;
147       //h=0.0001; t0=0.0; tfinal=0.01; t=t0;
148       n = ceil((tfinal-t0)/h)+1;        // number of timesteps
149       if(sw_method==3){ n=100;}
150       if(rank==0) printf("Params: h=%f, t0=%f, tfinal=%f, n=%d\n", h, t0, tfinal, n);
151    }
152
153    //if(h >= pow(delta_x,2)/8){if(rank==0){printf("ERROR: h >= pow(delta_x,2)/8 (%e >= %e ),
154    //              EXIT\n", h, pow(delta_x,2)/8);} error_flag++;}
155
156    if(error_flag>0){if(rank==0) printf("Errors = %d\n EXITING\n", error_flag); exit(0);}
157
158    junk = init_local_ranks();
159    junk = init_block_data();
160
161    return 0;
162 }
163
164 int init_local_ranks(){
165    // if(rank==0) printf("nprocs x y %d %d %d\n", nprocs, nprocsx, nprocsy);
166
167    // x, y rank location of current process
168    rank_x = rank % nprocsx;   // rank in x direction (mod)
169    rank_y = rank / nprocsx;   // rank in y direction (int division)
```

```
170
171     /* Ranks (blocks) go in this order
172        6 7 8
173        3 4 5
174        0 1 2  */
175
176     // ranks of neighboring processors
177     rank_down  = (rank_y-1)*nprocsx + rank_x  ;    if(rank_y  ==  0       ) rank_down  = -1;
178     rank_up    = (rank_y+1)*nprocsx + rank_x  ;    if(rank_y  == nprocsy-1) rank_up    = -1;
179     rank_left  =  rank_y   *nprocsx + rank_x-1;    if(rank_x  ==  0       ) rank_left  = -1;
180     rank_right =  rank_y   *nprocsx + rank_x+1;    if(rank_x  == nprocsx-1) rank_right = -1;
181
182     //printf("Ranks: me, x, y, up, down, left, right: %d %d %d %d %d %d %d \n", rank, rank_x,
183     //        rank_y, rank_up, rank_down, rank_left, rank_right);
184     return 0;
185  }
186
187  int init_block_data(){
188     int i, j;
189     double b, initx[nx+2], inity[nx+2], pi2;
190
191     /* Data goes in this order per block (border is ghost cells):
192        0 1 2 3 4 ... nx+1
193        1
194        2    ...
195        ...
196        ny+1             */
197
198     for (i=0; i<nx+2; i++){for (j=0; j<ny+2; j++){U1[i][j]=0;U[i][j]=0;}}  // init at 0
199
200     // initial conditions
201     if(rank== 0) {
202         i=10;j=10;U[i][j] = 0.1;
203         U[i-1][j]=U[i][j]/2;U[i+1][j]=U[i][j]/2;U[i][j-1]=U[i][j]/2;U[i][j+1]=U[i][j]/2;
204         U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;
205     }
206     if(rank== 0) {
207         i=60;j=30;U[i][j] =-0.2;
208         U[i-1][j]=U[i][j]/2;U[i+1][j]=U[i][j]/2;U[i][j-1]=U[i][j]/2;U[i][j+1]=U[i][j]/2;
209         U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;
210     }
211     if(rank== 8) {
212         i=30;j=30;U[i][j] = 1.0;
213         U[i-1][j]=U[i][j]/2;U[i+1][j]=U[i][j]/2;U[i][j-1]=U[i][j]/2;U[i][j+1]=U[i][j]/2;
214         U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;
215     }
216     if(rank== 6){
217         i=30;j=30;U[i][j] = 0.5;
218         U[i-1][j]=U[i][j]/2;U[i+1][j]=U[i][j]/2;U[i][j-1]=U[i][j]/2;U[i][j+1]=U[i][j]/2;
219         U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;
220     }
221     if(rank==14) {
222         i=30;j=30;U[i][j] =-1.0;
223         U[i-1][j]=U[i][j]/2;U[i+1][j]=U[i][j]/2;U[i][j-1]=U[i][j]/2;U[i][j+1]=U[i][j]/2;
224         U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;U[i-1][j-1]=U[i][j]/4;U[i+1][j+1]=U[i][j]/4;
225     }
226
227     if(sw_method==2){
228        // init ghost cell boundary conditions
229        b=0;
230        if(rank_down ==-1){j=ny+1; for (i=0; i<nx+2; i++){ U[i][j]=b;};};
231        if(rank_up   ==-1){j=0;    for (i=0; i<nx+2; i++){ U[i][j]=b;};};
232        if(rank_left ==-1){i=0;    for (j=0; j<ny+2; j++){ U[i][j]=b;};};
233        if(rank_right==-1){i=nx+1; for (j=0; j<ny+2; j++){ U[i][j]=b;};};
234     }
235     //fflush(stdout);MPI_Barrier(MPI_COMM_WORLD);
236     return 0;
237  }
238
239  /*******************************************************************************/
240  /* euler  ****************************************/
241  int euler(){
242     int i, j;
243     int i_1, i_nx, j_1, j_ny;            // boundary indicies
244
245     junk = laplacian();
246
247
248     for (i=1; i<nx+1; i++){
249        for (j=1; j<ny+1; j++){
250           U1[i][j] = U[i][j];    // progress our U's
251
252           // U[i][j] = U[i][j] + h*lu[i][j];  // Euler
253           U[i][j] = lu[i][j];  // simple scheme
254        }
255     }
256
```

```
257      return 0;
258   }
259
260   /******************************************************************************/
261   /* laplacian ****************************************/
262   int laplacian(){
263      int i, j;
264      double a=1;  // a is supposed to be the speed of wave propagation
265      double term1, term2, term3;
266
267      junk = ghost_cell_update(); // update ghost cells
268
269      if(sw_method==2){
270         for (i=1; i<nx+1; i++){
271            for (j=1; j<ny+1; j++){
272               if(t>0)
273               {  // simple scheme
274                  term1 = U[i-1][j] - 2*U[i][j] + U[i+1][j];
275                  term2 = U[i][j-1] - 2*U[i][j] + U[i][j+1];
276                  term3 = 2*U[i][j]-U1[i][j];
277                  lu[i][j] = pow(a,2)*pow(h,2)*(term1/pow(delta_x,2)+term2/pow(delta_y,2))+term3;
278               } else
279               { // initial time step
280                  term1 = U[i-1][j] - U[i+1][j];
281                  term2 = U[i][j-1] - U[i][j+1];
282                  term3 = U[i][j];
283                  lu[i][j] = a*h*(term1/delta_x+term2/delta_y)+term3;
284               }
285
286            }
287         }
288      }
289      return 0;
290   }
291
292   /******************************************************************************/
293   /* ghost_cell_update ****************************************/
294   int ghost_cell_update(){
295      int i, j;
296      int ierr, ierr1, ierr2, ierr3, tag=0;
297      MPI_Status status_rreq[4], status_sreq[4];
298      MPI_Request rreq[4], sreq[4];  // receive and send requests (down, up, left, right)
299      // receive and send buffers
300      double bufrd[nx],bufsd[nx],bufru[nx],bufsu[nx],bufrl[ny],bufsl[ny],bufrr[ny],bufsr[ny];
301
302      // DOWN **********************
303      if(rank_down==-1){ // down boundary condition;
304        //j=ny; for (i=1; i<=nx; i++){U[i][j]=U[i][j+1];}
305         sreq[0] = MPI_SUCCESS; rreq[0] = MPI_SUCCESS;
306      } else { // MPI
307         j=ny; for (i=1; i<nx+1; i++){ bufsd[i-1]=U[i][j];}
308         ierr2 = MPI_Isend(&bufsd, nx, MPI_DOUBLE, rank_down, tag, MPI_COMM_WORLD, &sreq[0]);
309         if (ierr2 != MPI_SUCCESS) {
310              printf("MPI_Isend ERROR status %d.\n", ierr2); ierr2=MPI_SUCCESS;
311         }
312         ierr1 = MPI_Irecv(&bufrd, nx, MPI_DOUBLE, rank_down, tag, MPI_COMM_WORLD, &rreq[0]);
313         if (ierr1 != MPI_SUCCESS) {
314              printf("MPI_Irecv ERROR status %d.\n", ierr1); ierr1=MPI_SUCCESS;
315         }
316      };
317
318      // UP **********************
319      if(rank_up==-1){ // up boundary condition;
320        //j=1; for (i=1; i<=nx; i++){U[i][j]=U[i][j-1];}
321         sreq[1] = MPI_SUCCESS; rreq[1] = MPI_SUCCESS;
322      } else { // MPI
323         j=1; for (i=1; i<nx+1; i++){ bufsu[i-1]=U[i][j];}
324         ierr2 = MPI_Isend(&bufsu, nx, MPI_DOUBLE, rank_up, tag, MPI_COMM_WORLD, &sreq[1]);
325         if (ierr2 != MPI_SUCCESS) {
326              printf("MPI_Isend ERROR status %d.\n", ierr2); ierr2=MPI_SUCCESS;
327         }
328         ierr1 = MPI_Irecv(&bufru, nx, MPI_DOUBLE, rank_up, tag, MPI_COMM_WORLD, &rreq[1]);
329         if (ierr1 != MPI_SUCCESS) {
330              printf("MPI_Irecv ERROR status %d.\n", ierr1); ierr1=MPI_SUCCESS;
331         }
332      };
333
334      // LEFT **********************
335      if(rank_left==-1){ // left boundary condition;
336        //i=1; for (j=1; j<=ny; j++){U[i][j]=U[i-1][j];}
337         sreq[2] = MPI_SUCCESS; rreq[2] = MPI_SUCCESS;
338      } else { // MPI
339         i=1; for (j=1; j<ny+1; j++){ bufsl[j-1]=U[i][j];}
340         ierr2 = MPI_Isend(&bufsl, ny, MPI_DOUBLE, rank_left, tag, MPI_COMM_WORLD, &sreq[2]);
341         if (ierr2 != MPI_SUCCESS) {
342              printf("MPI_Isend ERROR status %d.\n", ierr2); ierr2=MPI_SUCCESS;
343         }
```

```
344         ierr1 = MPI_Irecv(&bufrl, ny, MPI_DOUBLE, rank_left, tag, MPI_COMM_WORLD, &rreq[2]);
345         if (ierr1 != MPI_SUCCESS) {
346             printf("MPI_Irecv ERROR status %d.\n", ierr1); ierr1=MPI_SUCCESS;
347         }
348      };
349
350      // RIGHT **********************
351      if(rank_right==-1){ // right boundary condition;
352        //i=nx; for (j=1; j<=ny; j++){U[i][j]=U[i+1][j];}
353         sreq[3] = MPI_SUCCESS; rreq[3] = MPI_SUCCESS;
354      } else { // MPI
355         i=nx; for (j=1; j<ny+1; j++){ bufsr[j-1]=U[i][j];}
356         ierr2 = MPI_Isend(&bufsr, ny, MPI_DOUBLE, rank_right, tag, MPI_COMM_WORLD, &sreq[3]);
357         if (ierr2 != MPI_SUCCESS) {
358             printf("MPI_Isend ERROR status %d.\n", ierr2); ierr2=MPI_SUCCESS;
359         }
360         ierr1 = MPI_Irecv(&bufrr, ny, MPI_DOUBLE, rank_right, tag, MPI_COMM_WORLD, &rreq[3]);
361         if (ierr1 != MPI_SUCCESS) {
362             printf("MPI_Irecv ERROR status %d.\n", ierr1); ierr1=MPI_SUCCESS;
363         }
364      };
365
366
367      // update from receives
368      ierr3 = MPI_Waitall(4, rreq, MPI_STATUSES_IGNORE);
369      if (ierr3 != MPI_SUCCESS) {
370          printf("MPI_Waitall ERROR status %d.\n", ierr3); ierr3=MPI_SUCCESS;
371      }
372      if(rank_down  != -1){j=ny+1;for (i=1; i<nx+1; i++){U[i][j]=bufrd[i-1];}};
373      if(rank_up    != -1){j=0   ;for (i=1; i<nx+1; i++){U[i][j]=bufru[i-1];}};
374      if(rank_left  != -1){i=0   ;for (j=1; j<ny+1; j++){U[i][j]=bufrl[j-1];}};
375      if(rank_right != -1){i=nx+1;for (j=1; j<ny+1; j++){U[i][j]=bufrr[j-1];}};
376
377      // finish waiting for sends
378      ierr3 = MPI_Waitall(4, sreq, MPI_STATUSES_IGNORE);
379      if (ierr3 != MPI_SUCCESS) {
380          printf("MPI_Waitall ERROR status %d.\n", ierr3); ierr3=MPI_SUCCESS;
381      }
382
383      return 0;
384   }
385
386
387   int string2int(char *digit) {
388      int result = 0;
389      //--- Convert each digit char and add into result.
390      while (*digit >= '0' && *digit <='9') {
391         result = (result * 10) + (*digit - '0');
392         digit++;
393      }
394      //--- Check that there were no non-digits at end.
395      if (*digit != 0) {return -1;}
396      return result;
397   }
398
399
400   /*******************************************************************************/
401   /* output *****************************************/
402   int output(int nx,int ny,double x[],double y[],double u[nx][ny],double time) {
403
404     double xout[nx],yout[ny],uout[nx][ny];
405     double uout2[ny][nx];
406     double tmp;
407     int ierr,amode,rank,nproc,p,i,j;
408     MPI_File fh;
409     static int firstcall=1;
410     char *fname="wave.out";
411
412     MPI_Comm_rank(MPI_COMM_WORLD,&rank);
413     MPI_Comm_size(MPI_COMM_WORLD,&nproc);
414
415     if(rank==0) printf("Write: %s at t=%f\n",fname,time);
416
417     if (firstcall) {
418       amode = MPI_MODE_WRONLY | MPI_MODE_CREATE;
419       MPI_File_delete(fname,MPI_INFO_NULL);
420       firstcall=0;
421     }else{
422       amode = MPI_MODE_WRONLY | MPI_MODE_APPEND;
423     }
424     MPI_File_open(MPI_COMM_WORLD,fname,amode,MPI_INFO_NULL,&fh);
425
426     if (rank==0) {
427       double tmp=nproc;
428       MPI_File_write(fh,&tmp,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
429       tmp=nx;
430       MPI_File_write(fh,&tmp,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
```

```
431       tmp=ny;
432       MPI_File_write(fh,&tmp,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
433     }
434
435
436     for (p=0; p<nproc; ++p) {
437       if (rank==p) {
438         if (p==0)  {
439           memcpy(uout,u,8*nx*ny);
440           memcpy(xout,x,8*nx);
441           memcpy(yout,y,8*ny);
442         }else{
443           MPI_Send(x,nx,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
444           MPI_Send(y,ny,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
445           MPI_Send(u,nx*ny,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
446         }
447       }
448       if (rank==0) {
449         if (p!=0) {
450           MPI_Recv(xout,nx,MPI_DOUBLE,p,0,MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
451           MPI_Recv(yout,ny,MPI_DOUBLE,p,0,MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
452           MPI_Recv(uout,nx*ny,MPI_DOUBLE,p,0,MPI_COMM_WORLD,MPI_STATUSES_IGNORE);
453         }
454         // take the transpose of uout:
455         for (i=0; i<nx; ++i) for (j=0; j<ny; ++j ) uout2[j][i]=uout[i][j];
456         // output arrays:
457         MPI_File_write(fh,xout,nx,MPI_DOUBLE,MPI_STATUSES_IGNORE);
458         MPI_File_write(fh,yout,ny,MPI_DOUBLE,MPI_STATUSES_IGNORE);
459         MPI_File_write(fh,uout2,nx*ny,MPI_DOUBLE,MPI_STATUSES_IGNORE);
460       }
461     }
462
463     if (rank==0)
464      MPI_File_write(fh,&time,1,MPI_DOUBLE,MPI_STATUSES_IGNORE);
465
466     MPI_File_close(&fh);
467     return 0;
468   }
469
470   /* EOF */
```

Listing 2: Matlab script `wave.m` used to generate wave plots.

```
1   clear
2   fname='wave.out';
3   fid=fopen(fname,'r');
4   colormap(gray);
5
6   sw = 2; % 2=top-down, 3=3D
7
8   while (fid ~= 1)
9           [nprocs,count]=fread(fid,1,'float64');
10          if (count~=1) break; end;
11          [nx,count]=fread(fid,1,'float64');
12          if (count~=1) break; end;
13          [ny,count]=fread(fid,1,'float64');
14          if (count~=1) break; end;
15
16      if sw==2; figure(1);clf;hold on; end;           % for top-down surfl
17          for i=1:nprocs
18                  [x,count]=fread(fid,nx,'float64');
19                  [y,count]=fread(fid,ny,'float64');
20                  [u,count]=fread(fid,[nx,ny],'float64');
21                  u=u';
22                  disp(sprintf('contouring block %i / %i',i,nprocs))
23                  v=-1:.05:1;
24                  surfl(x,y,u);
25                  %contour(x,y,u,v);
26                  %contour(x,y,u);
27         if sw==3; hold on; end;                      % for 3D surfl
28                  axis equal;
29                  axis([0.0, 1.0, 0.0, 1.0]);
30                  %pause(.001)
31          end
32          shading interp
33          hold off;
34
35          [time,count]=fread(fid,1,'float64');
36          title(['t=', num2str(time)])
37
38      if sw==2;
39         plot_name = strcat('ppproj3_waves_t',num2str(100*time),'.eps'); % plot name
40         print(gcf, '-depsc2', plot_name);            % print plot
41      end;
42      if sw==3;
43         plot_name = strcat('ppproj3_waves3d_t',num2str(100*time),'.eps'); % plot name
44         print(gcf, '-depsc2', plot_name);            % print plot
45      end;
46
47          disp(sprintf('press a key to continue... done plotting solution at t=%f',time));
48          pause(.25)
49
50
51  end
52  fclose(fid);
53  disp('Error reading file, or reached end of file');
```

Listing 3: Matlab script `plots_a.m` used to generate time and scalability plots on Machine A.

```
function plots

    y_60 = [11.256163 2.97289 2.085062 1.533152 1.26716 0.996021 ];
    y_120 = [45.810661 11.491478 7.839005 5.383871 4.187208 3.232426 ];
    y_180 = [111.721949 26.280184 17.838946 12.176307 9.413069 7.149453 ];
    y_240 = [205.516207 48.729024 32.982407 22.048036 17.014949 13.091097 ];
    y_300 = [324.565092 80.836117 52.27633 35.216773 27.207582 20.906174 ];
    y_360 = [469.384824 121.780736 79.046601 52.920637 39.762668 30.649895 ];
    y_420 = [640.612062 169.099137 111.880208 73.514546 55.915544 42.594644 ];
    y_480 = [837.288399 222.474732 148.684467 98.82164 74.699427 56.220476 ];
    y_540 = [1058.459214 282.420194 189.511962 127.652788 96.424833 72.806032 ];
    y_600 = [1303.937413 349.278586 234.46219 159.520055 121.226566 91.406496 ];
    y_660 = [1572.888626 425.870722 283.411603 194.181991 148.103974 112.493721 ];
    y_720 = [1864.402242 503.134313 336.78972 231.952082 176.332697 135.101946 ];
    y_780 = [2175.61912 591.267074 394.989867 271.357013 206.680045 158.929052 ];

    procs = [1 4 6 9 12 16];
    ys = [y_60; y_120; y_180; y_240; y_300; y_360; y_420; y_480; y_540; y_600; y_660; ...
          y_720; y_780];
    labels ={'g = 60', 'g = 120', 'g = 180', 'g = 240', 'g = 300', 'g = 360', 'g = 420', ...
             'g = 480', 'g = 540', 'g = 600', 'g = 660', 'g = 720', 'g = 780'};

    figure(1);
    semilogy(procs,ys,'o-');
    box('on');
    grid('on');
    title('Execution Time');
    ylabel('Time (s)');
    xlabel('Processors');
    %legend(labels);
    plot_name = strcat('execution_time_semilogy.eps'); % plot name
    print(gcf, '-depsc2', plot_name);                  % print plot

    figure(2);
    plot(procs,ys,'o-');
    box('on');
    grid('on');
    title('Execution Time');
    ylabel('Time (s)');
    xlabel('Processors');
    legend(labels);
    plot_name = strcat('execution_time_linear.eps'); % plot name
    print(gcf, '-depsc2', plot_name);                % print plot

    y_60_s  = speedup(y_60);   y_60_e  = efficiency(y_60, procs);
    y_120_s = speedup(y_120);  y_120_e = efficiency(y_120, procs);
    y_180_s = speedup(y_180);  y_180_e = efficiency(y_180, procs);
    y_240_s = speedup(y_240);  y_240_e = efficiency(y_240, procs);
    y_300_s = speedup(y_300);  y_300_e = efficiency(y_300, procs);
    y_360_s = speedup(y_360);  y_360_e = efficiency(y_360, procs);
    y_420_s = speedup(y_420);  y_420_e = efficiency(y_420, procs);
    y_480_s = speedup(y_480);  y_480_e = efficiency(y_480, procs);
    y_540_s = speedup(y_540);  y_540_e = efficiency(y_540, procs);
    y_600_s = speedup(y_600);  y_600_e = efficiency(y_600, procs);
    y_660_s = speedup(y_660);  y_660_e = efficiency(y_660, procs);
    y_720_s = speedup(y_720);  y_720_e = efficiency(y_720, procs);
    y_780_s = speedup(y_780);  y_780_e = efficiency(y_780, procs);

    figure(3);
    yss = [y_60_s; y_120_s; y_180_s; y_240_s; y_300_s; y_360_s; y_420_s; y_480_s; ...
           y_540_s; y_600_s; y_660_s; y_720_s; y_780_s];
    hold('on')
    plot(procs,yss,'o-');
    plot(procs,procs,'-.');
    box('on');
    grid('on');
    title('Parallel Speedup');
    ylabel('Speedup relative to one processor (Ts/Tp)');
    xlabel('Processors');
    legend(labels, 'Location', 'NorthWest');
    plot_name = strcat('speedup_4.eps'); % plot name
    print(gcf, '-depsc2', plot_name);    % print plot

    figure(4);
    yss_e = [y_60_e; y_120_e; y_180_e; y_240_e; y_300_e; y_360_e; y_420_e; y_480_e; ...
             y_540_e; y_600_e; y_660_e; y_720_e; y_780_e];
    plot(procs,yss_e,'o-');
    title('Parallel Efficiency');
    ylabel('Efficiency (Ts/(p*Tp))');
    xlabel('Processors');
    %legend(labels);
    grid;
    plot_name = strcat('parallel_efficiency.eps'); % plot name
    print(gcf, '-depsc2', plot_name);    % print plot
```

```
85
86      figure(5);
87
88      subplot(2,2,1);
89      semilogy(procs,ys,'o-');
90      title('Execution Time');
91      ylabel('Time (s)');
92      xlabel('Processors');
93      grid;
94
95      subplot(2,2,2);
96      plot(procs,ys,'o-');
97      title('Execution Time');
98      ylabel('Time (s)');
99      xlabel('Processors');
100     grid;
101
102     subplot(2,2,3);
103     hold('on');
104     plot(procs,yss,'o-');
105     plot(procs,procs,'-.');
106     title('Parallel Speedup');
107     ylabel('Speedup relative to one processor');
108     xlabel('Processors');
109     hold('off');
110     grid('on');
111     box('on');
112
113     subplot(2,2,4);
114     plot(procs,yss_e,'o-');
115     title('Parallel Efficiency');
116     ylabel('Efficiency');
117     xlabel('Processors');
118     grid('on');
119 end
120
121 function e = efficiency(time, procs)
122     [x, y] = size(time);
123     e = zeros(1, y);
124     for i=1:y
125         e(i) = time(1)/(procs(i) * time(i));
126     end;
127 end
128
129 function s = speedup(time)
130     [x, y] = size(time);
131     s = zeros(1, y);
132     for i=1:y
133         s(i) = time(1)/time(i);
134     end;
135 end
```

Listing 4: Matlab script `plots_b.m` used to generate time and scalability plots on Machine B.

```
function plots
    y_60 = [7.38137 2.536994 1.600167 1.252972 1.037541 0.802509 0.679604 0.669928 ...
            0.629778 0.562864 0.574864 0.54802 0.541736 0.528685 0.525228 0.527546 ];
    y_120 = [30.600927 10.31687 6.825848 4.82463 3.483436 2.762999 2.280772        ...
             1.910099 1.709575 1.419545 1.118151 0.9641 0.875441 0.637029 0.598223 ...
             0.567635 ];
    y_180 = [77.327193 24.564606 16.028164 11.403042 8.387821 6.781668 5.815331    ...
             4.875607 4.294808 3.580791 3.359725 2.929225 2.504102 1.766428        ...
             1.587851 1.395773 ];
    y_240 = [146.442535 44.497967 31.810037 21.406071 16.591471 13.635886          ...
             11.499007 9.584617 8.53982 7.303088 7.395752 6.157588 5.337012        ...
             3.656079 3.313398 2.850585 ];
    y_300 = [239.46117 75.388873 51.078114 35.937671 27.471216 21.430344           ...
             18.925514 16.188306 14.405945 12.717983 12.639934 10.80496 9.33495    ...
             6.413734 5.718 4.990584 ];
    y_360 = [354.532545 112.400235 75.146513 54.716719 42.968344 33.800313         ...
             29.275931 24.791546 21.704155 19.32376 19.86955 16.661621 14.489097   ...
             10.055773 9.149919 7.871039 ];
    y_420 = [493.782822 151.351127 104.969557 76.897367 58.177394 45.881191        ...
             41.176187 34.557753 31.224206 27.553418 27.900372 23.92885 20.836552  ...
             14.507633 13.075257 11.43468 ];
    y_480 = [652.169093 200.065266 133.835009 101.915608 81.343607 62.03498        ...
             53.774262 46.468106 41.682039 37.382322 38.099407 32.119105 28.049637 ...
             19.722225 17.997581 15.75444 ];
    y_540 = [830.788974 236.995249 171.521422 128.17276 101.841207 79.998566       ...
             69.94026 59.196067 53.92978 47.853129 48.48761 41.085497 36.11513     ...
             25.300582 22.588998 19.851638 ];
    y_600 = [1015.109329 275.422983 203.195084 159.261321 128.774914 103.392809    ...
             87.663516 74.6749 67.196406 59.223197 61.169977 51.303099 44.364831   ...
             30.04596 26.827983 23.650394 ];
    y_660 = [1226.055597 365.042804 253.855654 196.014785 154.911348 121.902304    ...
             105.036497 89.614245 79.13965 69.552028 72.831975 59.886399 51.206887 ...
             34.745652 30.774395 27.167085 ];
    y_720 = [1447.591597 421.376879 294.105125 225.66063 182.703805 142.624019     ...
             122.32387 103.439811 91.573362 80.127017 83.825005 68.27377 58.734826 ...
             39.837584 35.495454 31.269871 ];
    y_780 = [1666.362467 489.863741 353.168962 263.599598 209.035117 163.505753    ...
             141.020763 117.902523 105.16773 91.876026 94.722774 77.560809 66.3928 ...
             44.790802 39.942517 35.072835 ];

    procs = [1 4 6 9 12 16 20 25 30 36 40 50 60 100 120 144];
    ys = [y_60; y_120; y_180; y_240; y_300; y_360; y_420; y_480; y_540; y_600; y_660; ...
          y_720; y_780];
    labels ={'g = 60', 'g = 120', 'g = 180', 'g = 240', 'g = 300', 'g = 360', 'g = 420', ...
             'g = 480', 'g = 540', 'g = 600', 'g = 660', 'g = 720', 'g = 780'};

    figure(1);
    semilogy(procs,ys,'o-');
    box('on');
    grid('on');
    title('Execution Time');
    ylabel('Time (s)');
    xlabel('Processors');
    %legend(labels);
    plot_name = strcat('execution_time_semilogy.eps'); % plot name
    print(gcf, '-depsc2', plot_name);                  % print plot

    figure(2);
    plot(procs,ys,'o-');
    box('on');
    grid('on');
    title('Execution Time');
    ylabel('Time (s)');
    xlabel('Processors');
    legend(labels);
    plot_name = strcat('execution_time_linear.eps'); % plot name
    print(gcf, '-depsc2', plot_name);                % print plot

    y_60_s  = speedup(y_60);   y_60_e  = efficiency(y_60, procs);
    y_120_s = speedup(y_120);  y_120_e = efficiency(y_120, procs);
    y_180_s = speedup(y_180);  y_180_e = efficiency(y_180, procs);
    y_240_s = speedup(y_240);  y_240_e = efficiency(y_240, procs);
    y_300_s = speedup(y_300);  y_300_e = efficiency(y_300, procs);
    y_360_s = speedup(y_360);  y_360_e = efficiency(y_360, procs);
    y_420_s = speedup(y_420);  y_420_e = efficiency(y_420, procs);
    y_480_s = speedup(y_480);  y_480_e = efficiency(y_480, procs);
    y_540_s = speedup(y_540);  y_540_e = efficiency(y_540, procs);
    y_600_s = speedup(y_600);  y_600_e = efficiency(y_600, procs);
    y_660_s = speedup(y_660);  y_660_e = efficiency(y_660, procs);
    y_720_s = speedup(y_720);  y_720_e = efficiency(y_720, procs);
    y_780_s = speedup(y_780);  y_780_e = efficiency(y_780, procs);

    figure(3);
    yss = [y_60_s; y_120_s; y_180_s; y_240_s; y_300_s; y_360_s; y_420_s; y_480_s; ...
```

```
85              y_540_s; y_600_s; y_660_s; y_720_s; y_780_s];
86      hold('on')
87      plot(procs,yss,'o-');
88      plot(procs,procs,'-.');
89      box('on');
90      grid('on');
91      title('Parallel Speedup');
92      ylabel('Speedup relative to one processor (Ts/Tp)');
93      xlabel('Processors');
94      legend(labels, 'Location', 'NorthWest');
95      plot_name = strcat('speedup.eps'); % plot name
96      print(gcf, '-depsc2', plot_name);    % print plot
97
98      figure(4);
99      yss_e = [y_60_e; y_120_e; y_180_e; y_240_e; y_300_e; y_360_e; y_420_e; y_480_e; ...
100               y_540_e; y_600_e; y_660_e; y_720_e; y_780_e];
101     plot(procs,yss_e,'o-');
102     title('Parallel Efficiency');
103     ylabel('Efficiency (Ts/(p*Tp))');
104     xlabel('Processors');
105     %legend(labels);
106     grid;
107     plot_name = strcat('parallel_efficiency.eps'); % plot name
108     print(gcf, '-depsc2', plot_name);              % print plot
109
110     figure(5);
111
112     subplot(2,2,1);
113     semilogy(procs,ys,'o-');
114     title('Execution Time');
115     ylabel('Time (s)');
116     xlabel('Processors');
117     grid;
118
119     subplot(2,2,2);
120     plot(procs,ys,'o-');
121     title('Execution Time');
122     ylabel('Time (s)');
123     xlabel('Processors');
124     grid;
125
126     subplot(2,2,3);
127     hold('on');
128     plot(procs,yss,'o-');
129     plot(procs,procs,'-.');
130     title('Parallel Speedup');
131     ylabel('Speedup relative to one processor');
132     xlabel('Processors');
133     hold('off');
134     grid('on');
135
136     subplot(2,2,4);
137     plot(procs,yss_e,'o-');
138     title('Parallel Efficiency');
139     ylabel('Efficiency');
140     xlabel('Processors');
141     grid('on');
142     plot_name = strcat('all.eps');    % plot name
143     print(gcf, '-depsc2', plot_name); % print plot
144 end
145
146 function e = efficiency(time, procs)
147     [x, y] = size(time);
148     e = zeros(1, y);
149     for i=1:y
150         e(i) = time(1)/(procs(i) * time(i));
151     end;
152 end
153
154 function s = speedup(time)
155     [x, y] = size(time);
156     s = zeros(1, y);
157     for i=1:y
158         s(i) = time(1)/time(i);
159     end;
160 end
```