AIAA 2000-0346

Self-Scheduling Parallel Methods for Multiple Serial Codes with Application to WOPWOP

Lyle N. Long
The Pennsylvania State University
University Park, PA

Kenneth S. Brentner
NASA Langley Research Center
Hampton, VA

38th Aerospace Sciences Meeting & Exhibit
10 – 13 January 2000 / Reno, NV

Self-Scheduling Parallel Methods for Multiple Serial Codes with Application to WOPWOP

Lyle N. Long*
The Pennsylvania State University
University Park, PA 16802
lnl@psu.edu

and

Kenneth S. Brentner†
NASA Langley Research Center
Hampton, VA 23681
k.s.brentner@larc.nasa.gov

ABSTRACT

This paper presents a scheme for efficiently running a large number of serial jobs on parallel computers. Two examples are given of computer programs that run relatively quickly, but often they must be run numerous times to obtain all the results needed. It is very common in science and engineering to have codes that are not massive computing challenges in themselves, but due to the number of instances that must be run, they do become large-scale computing problems. The two examples given here represent common problems in aerospace engineering: aerodynamic panel methods and aeroacoustic integral methods. The first example simply solves many systems of linear equations. This is representative of an aerodynamic panel code where someone would like to solve for numerous angles of attack. The complete code for this first example is included in the appendix so that it can be readily used by others as a template. The second example is an aeroacoustics code (WOPWOP) that solves the Ffowcs Williams–Hawkings equation to predict the far-field sound due to rotating blades. In this example, one quite often needs to compute the sound at numerous observer locations, hence parallelization is utilized to automate the noise computation for a large number of observers.

INTRODUCTION

The purpose of this paper is to illustrate how one can use a wide variety of parallel machines (e.g. parallel supercomputers, Beowulf clusters, or a heterogeneous workstation cluster) to solve a large number of small problems. With the rapid increase in computer speed, more and more scientists and engineers are content with the computational performance of workstations. Workstations can now solve computational fluid dynamics problems consisting of millions of grid points, and many people are content with that many degrees of freedom. While it is crucial to have some people 'pushing the envelope' and solving problems with billions of grid points, fewer and fewer people need (or can afford) the power offered by the large machines found at national supercomputer centers. In addition, the supercomputing community often over-emphasizes massive grand challenge type problems. However, there are a huge number of important engineering and scientific problems that involve relatively small codes that run relatively quickly. These codes could take advantage of supercomputer centers and networked workstations by using the methodology proposed here.

For example, there are some very important scientific questions in surface chemistry[1], supercritical fluids[2,3], quantum Monte Carlo[4], chemical kinetics[5], and aeroacoustics[6] that can be tackled using relatively small-scale molecular dynamics or Monte Carlo codes. However, one often needs to run these codes repeatedly to either build up statistical data or vary input parameters. As this paper will show, one can very easily use the Message Passing Interface (MPI) paradigm combined with the Fortran or C programming languages, to run thousands of different cases in a very short amount of time.

We discuss two different applications here. The first code solves a large number of linear equations while the second code (WOPWOP) solves an aeroacoustics integral equation.

MULTIPLE GAUSSIAN ELIMINATION CODES

Before we describe the self-scheduling version of the WOPWOP aeroacoustics code, we will present a much simpler program, which solves numerous NxN linear systems of equations on P processors. Each processor creates a matrix **A** of random numbers and a right hand side (RHS) of random numbers, and solves the system. One could easily change this code so that each processor is sent a matrix and a RHS, or each processor could use the same matrix and be sent a different RHS (This scenario is representative of an aerodynamic panel code or an electromagnetic moment method code).

* Professor, Department of Aerospace Engineering, Associate Fellow
† Senior Research Engineer, Computational Modeling and Simulation Branch, Senior Member

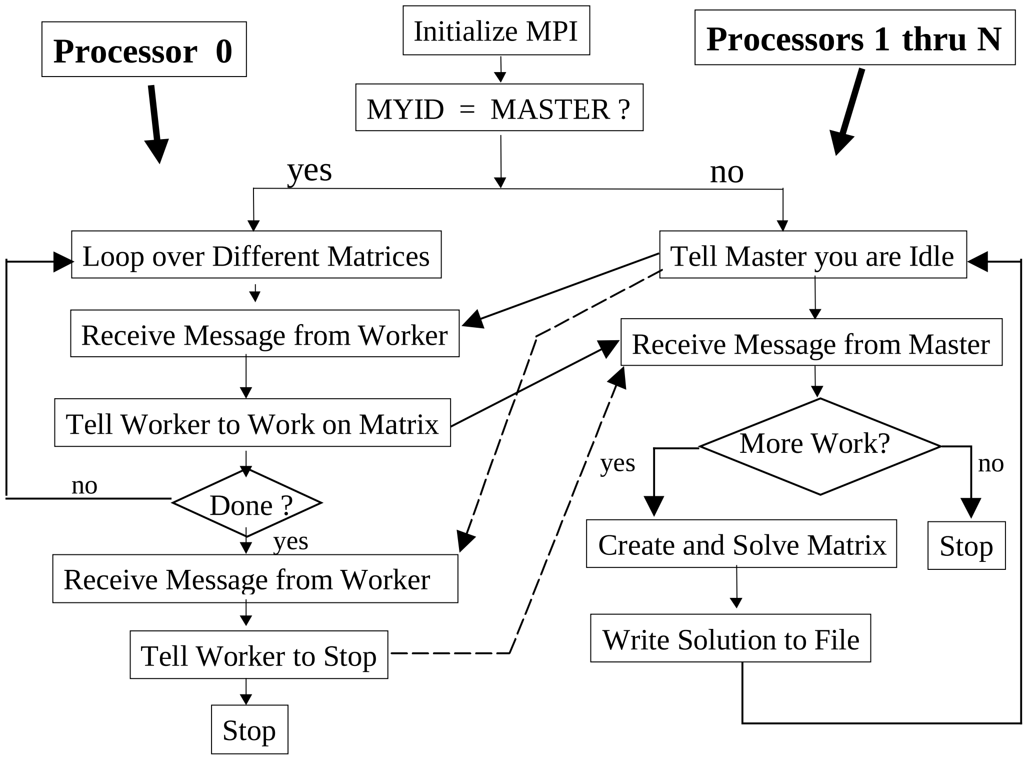

Figure 1. Schematic of the self-scheduling matrix solver code.

The self-scheduling approach presented in this paper is based upon the Single Program Multiple Data (SPMD) parallel programming paradigm. In order to understand this paradigm, you need to understand that every processor on the parallel computer (or workstation cluster) will run the same computer program. Each processor may execute different portions of the code or use different input data however, and this is accomplished primarily by using Fortran IF statements that use the variable 'myid', which is a unique number that each processor will have assigned to it. References 7 and 8 describe this in more detail.

Figure 1 shows a flowchart of this program. The code has one large IF–THEN–ELSE construct. The master processor performs the first half of IF–THEN–ELSE, while the slaves perform the second half (the ELSE portion). The code that runs on the master has a DO–LOOP that loops over the number of different linear systems of equations to solve. For each iteration of the DO–LOOP, the master processor waits to get a message from an idle slave processor, and then tells the slave to work on a new matrix. Once the master has finished the DO–LOOP, it then loops through all the slave processors, and tells each one to stop working. This insures that the program ends smoothly. You would not want to have the master stop before telling all the slaves to stop working.

At the start of the execution, each of the slave processors first sends a message to the master notifying the master that they are ready for work. The slave processors then wait to get a message from the master. Once the message is received, it checks to see if the master wants it to solve another system of equations, or stop working.

The only MPI routines used by this program are:

MPI_INIT	Initializes MPI
MPI_COMM_RANK	Each processor finds it's Processor ID
MPI_COMM_SIZE	Determine how many processors there are
MPI_ABORT	In case of error, aborts the job
MPI_SEND	Sends a message from one processor to another
MPI_RECV	Receives a message from one processor to another
MPI_WTIME	Computes wall clock time
MPI_FINALIZE	Stops MPI

While MPI has over a hundred different functions, only eight of them are required here (actually, MPI_WTIME isn't essential). For more information on MPI, see Reference 9.

This program was able to create and solve 120 different systems of 3000x3000 linear equations in two hours on forty-one 400 MHz Pentium II processors connected via a fast-ethernet switch. So one could run roughly 120 simultaneous copies of an aerodynamic panel method in nearly the same time. Using a single PC, this same amount of work would take 3.5 days. This approach has a parallel efficiency of essentially 100%. Of course there are other ways to run codes such as this in parallel (e.g. OpenMP, Python, etc.), but for the time being MPI is a standard, portable way to do parallel programming.

The complete F90/MPI code (named PGAUSS[10]) is included in the Appendix for completeness. This code is only intended as an example, not a production code. The basic framework of this code could be used for a large variety of problems, for example Monte Carlo chemistry codes, molecular dynamics, aerodynamic panel methods, or optimization codes. Any program that needs to be run hundreds or thousands of times, and each run requires a moderate number of floating point operations. All one would have to do is replace the box in the flowchart labeled 'Create and Solve Matrix' with the new application. In the program, all that is needed is to call the new function instead of Gauss.

THE WOPWOP NOISE PREDICTION PROGRAM

The WOPWOP computer program[11], which is based upon the Ffowcs Williams–Hawkings (FW–H) equation[12], is widely used for rotorcraft noise predictions. The computer program solves an integral formulation for the noise, given the rotor blade motion and blade surface pressure. WOPWOP runs quite fast compared to computational fluid dynamics (CFD) methods, but one usually wants to run this program numerous times. It would be useful to be able to run hundreds or thousands of copies of the program very rapidly, and to have the output organized so that it can be post-processed easily. This section describes a notional parallel version of WOPWOP where N versions of WOPWOP can be run on P processors. This self-scheduling version will run on large parallel supercomputers or clusters of workstations. Results are presented for WOPWOP running on a Beowulf-class cluster of PC's running Linux[13], a Cray T3E, and an SGI Origin 2000.

WOPWOP[11] is a computer code that predicts the discrete frequency noise of conventional and advanced rotating blades. WOPWOP implements acoustic formulation 1A of Farassat[14]—a time-domain, integral representation of the Ffowcs Williams–Hawkings (FW–H) equation[12], which is valid in both the near and far field. The FW–H equation is the most general form of the Lighthill acoustic analogy[15] and is appropriate when sound is generated by surfaces in arbitrary motion. WOPWOP calculates the rotor noise by integrating over the actual surface of the rotor and uses the flapping, feathering, and lead-lag motions of the blade up to the second harmonic. WOPWOP can model either stationary or moving observers. A complete description of the WOPWOP code along with the description of inputs can be found in Reference 11.

The WOPWOP code is a relatively small, computationally efficient code that can compute the noise signal from a helicopter rotor in tens of seconds (in some cases) on a scientific workstation. Although this sounds like a small amount of time, it is not unusual to compute the noise at thousands of observer locations on a surface to characterize the noise directivity. For example, if the computation for a single observer takes only 30 seconds but there are 1024 observer locations, the total CPU time require will still be more than 8.5 CPU hours! In addition, some cases may require significant computer resources for each run (e.g. the noise due to a maneuvering helicopter). WOPWOP is a good candidate for self-scheduled parallelization because the program size is small and the noise computed at one observer location is completely independent from all other observer locations. Thus, the computation of noise for different observer locations can be shared across a larger number of processors with essentially no communication between processors.

SELF-SCHEDULING PARALLEL WOPWOP

In order to run hundreds or thousands of copies of WOPWOP relatively quickly, the above PGAUSS program has been converted to handle WOPWOP, instead of the Gaussian elimination program. The main program in WOPWOP was converted to a subroutine, and a new main program was written, which is very similar to that used in PGAUSS. Relatively few changes were made to WOPWOP. Only a few parameters are passed from the new WOPWOP main program[16] to the WOPWOP main subroutine. One of these parameters is the observer location, and the others are unique file names for each WOPWOP run. It is quite easy to revert to the serial version of WOPWOP simply by compiling the program with a different (and trivial) main program, so only one version of WOPWOP has to be maintained.

Organizing Input/Output Files

In order to make the entire process as efficient as possible, it is important to arrange the input and output well. At the present time, in the parallel version of the WOPWOP code each processor reads the same input file. This could be easily changed so that each one reads a different file, if that was useful. In addition, each processor currently writes an output file for each case it runs. Alternatively, one could have each processor send the final results back to the master, and the master could write out just one file. The ultimate input/output implementation depends somewhat on user preferences, but it is also important to ensure that communication costs are minimal and that the load is efficiently balanced (e.g., the master processor does not have too much work to do so that the slave processors have to wait unnecessarily).

For this paper, the output from each processor is directed to a new file each time a processor runs a new case. This is done using code such as:

```
write( myfile,'(a,i3.3,a)' ) &
    'wop', int(xnew(4)),'.out'
```

The variable 'myfile' is declared a character variable, and will be used to define a unique filename for each case run in WOPWOP. The variable xnew(4) is an integer corresponding to the case number being run. Therefore, if we were running 120 different observer locations, xnew(4) would vary from 1 to 120. So the output of case number 60 would be written to a file named 'wop060.out'. WOPWOP also writes out two other files for each case. One of these has the time history of the signal and the other has the overall sound pressure level (OASPL) data. For observer location 60, these would be called: wop060.dat and oaspl060.dat, respectively.

In this implementation, WOPMAIN is called with a parameter list of four variables. The first variable, 'xnew', is a real vector of length 4. This vector contains the x, y, and z offset utilized to determine observer location of this instance of execution. The fourth variable in 'xnew' is the case number.

The other three variables passed to WOPMAIN are the three file names (myfile, wopfile and splfile), of the output files described above. For each observer location WOPMAIN is called, it will read the input file, and then it will create three output files. So for a run with 120 cases, there would be 360 output files.

The implementation described here is just one example of how input and output can be handled. In future versions of WOPWOP, we will permit several different cases to be included in the namelist file simultaneously, but we will also keep the option of being able to specify an array of observer locations. We will also send some of the output data back to the main program, so that it can be organized into a single output file.

RESULTS

Figure 2 shows the results of running WOPWOP on three Pentium II processors, and using 64 observer locations. The results were easily used to create an 8x8 array of thickness noise for different observer locations. This run required 430 CPU seconds.

In addition to OASPL levels, WOPWOP also stores the time histories of the acoustic pressure for each observer location. Figure 3 shows an example of this, by presenting the time histories at points 1, 32 and, 64 from the case shown in Figure 2.

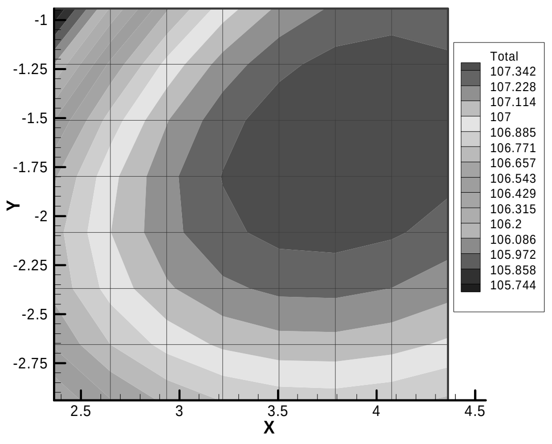

Figure 2. Contour plot of OASPL for different observer locations.

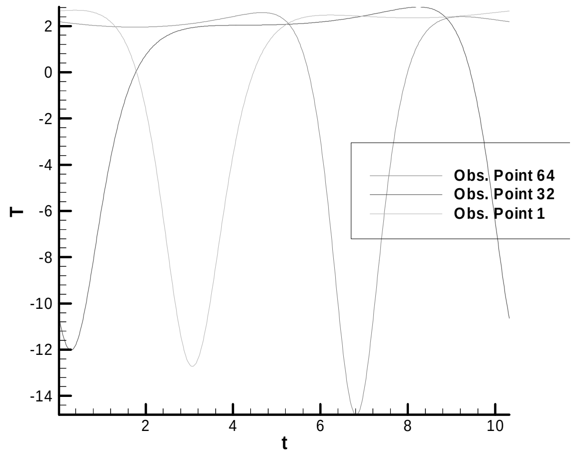

Figure 3. Time histories of pressure at three different observer locations.

Code Performance

In order to illustrate the speed of the parallel version of WOPWOP, Figure 4 shows the CPU time required to compute a time history with 512 points at 400 observer locations using different numbers of processors on the PC cluster. Since this approach is perfectly parallel (essentially no inter-processor communications are required), the computer time scales perfectly with number of processors. While it took 4800 seconds to run the 400 observer solutions on one processor (and one master processor), it took only 127 seconds using 47 processors (and one master processor).

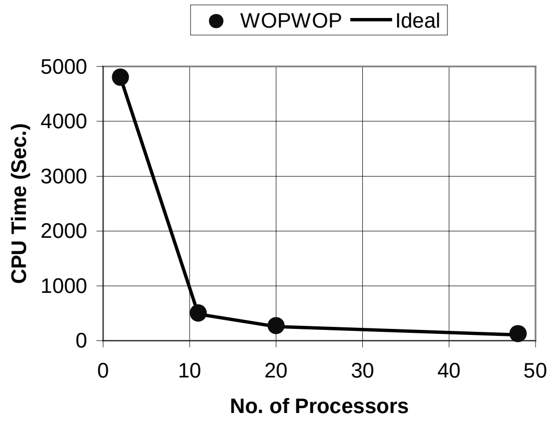

Figure 4. Time required to solve for 400 observer positions on the PC cluster.

The WOPWOP code was also run on several other parallel computers. In all these cases, 48 processors were used and 400 observer locations were computed. Table 1 shows the timing results for these runs. The results show that the PC clusters can be very effective computing platforms for problems such as these. It should be noted that the variables in the program are simply defined as 'REAL,' and the default precision was used on each machine (which is 64-bit on the T3E). However, the code was run on the SGI Origin 2000 with the –r8 option (which promotes all real variables to 64-bit precision automatically), and in this case the CPU time was 98 seconds. So precision had little effect on the timings (at least on the SGI Origin). In addition, it should be noted that the times shown below include time for both computation and file input and output. This same example was run on the Origin 2000 using 100 processors and it required 56 seconds.

Computer	Compile Options	Time (Sec.)
Beowulf Cluster, Penn State, 400 MHz PII	- O3	127
Cray T3E, NASA Goddard, 300 MHz Alpha 21164	- O3	177
SGI O2000 NCSA 195 MHz R10000	- O3	95

Table 1. CPU Times on Several Different Computers using 48 Processors and 400 Observer Locations.

In the present version of WOPWOP, each processor reads the input file for each case it runs. In some cases, such as when you have detailed surface loading data in the input file, the file could be large. In this situation, it may be undesirable to have each processor reading from the same file, hence a better alternative would then have the master processor read the file and then broadcast or send the data to the other processors. A typical case might require 80 chordwise, 30 spanwise, and 1440 azimuthal points with two floating point numbers required at each point. In single precision, this would amount to 28 megabytes (224 megabits) of data. Engineers are now generally aware of the approximation used to compute computational time,

$$\textit{CPU Time} = 10^{-6}\,\frac{\textit{Floating Point Operations}}{\textit{Megaflops}}$$

but they are less use to thinking of communication time. Nevertheless, a rough approximation of the same form is available to determine the communication time required. This approximation is written

$$\textit{Communication Time} = 10^{-6}\,\frac{\textit{Bits of Data Sent}}{\textit{Megabits / Second}}$$

The above estimate does not included the overhead (latency) in sending data., but it enables one to make crude estimates of the cost of sending data. On large parallel supercomputers, latency is measured in microseconds, while on Beowulf clusters it can be significantly larger. If a fast-ethernet network can sustain 50 megabits/second (50% of peak), then it might require roughly 4 seconds to send the 28 megabytes of data. For future versions of WOPWOP (or other self-scheduling parallel codes), it is important to be able to understand the cost of doing computations and the cost of communicating between processors as well so that appropriate code design choices can be made. It should be mentioned that there are ways to hide the cost of the communications, such as when your computer and algorithm allow you to communicate and compute simultaneously.

<u>BEOWULF PC CLUSTERS</u>

The COst effective COmputing Array (COCOA) at Penn State[17,18,19] is a 50-processor cluster of off-the-shelf PCs connected via fast ethernet (100 Mbit/sec). The PCs run RedHat Linux with MPI for parallel programming and DQS for queueing the jobs. Each node of COCOA consists of dual 400 MHz Intel Pentium II Processors in a symmetric-multiprocessor (SMP) configuration with 512 MB of memory (512 KB L2 cache). A single Baynetworks 27-port fast-ethernet switch with a backplane bandwidth of 2.5 Gigabits per second is used for the networking. The entire system cost approximately $100,000 (in 1998). Detailed information on how COCOA was built can be obtained from its web-site.[17] COCOA was built to enable the study of complex fluid dynamics problems using CFD without depending on expensive external supercomputing resources. This system could be easily expanded to include more processors. Two Fortran compilers, the Portland Group Fortran compiler and the Absoft Pro Fortran compiler, in addition to the GNU C (gcc) compiler have been utilized on COCOA.

In order to illustrate the computational speed of the machine, Figure 5 shows the Mflops obtained from an inviscid, unstructured CFD code.[18,19] For this case, an unstructured tetrahedral grid with 483,565 cells was used and the run consumed 1.2 GB of RAM. The benchmark showed that COCOA was almost twice as fast as an older IBM SP2 (66 MHz RS/6000-370 nodes) for this application. In order to show the networking speeds, Figure 6 shows the results obtained from the *netperf* test :

```
netperf –t UDP_STREAM -l 60 -H <target-machine>
-- -s 65535 -m <packet-size>
```

This is indicative of the communication speed between any two nodes. It is seen that almost 96% of the peak communication speed of 100 Mbit/sec is achieved between any two nodes for packet sizes above 1000 bytes.

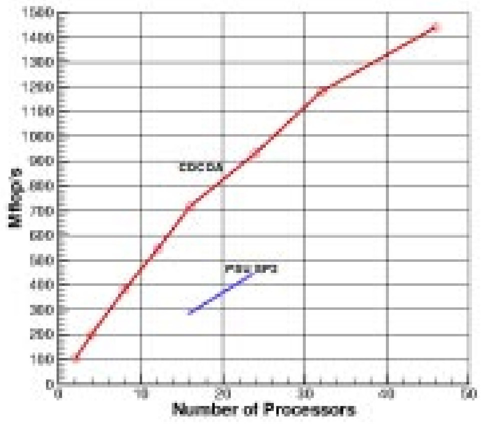

Figure 5. Timings for a CFD code run on COCOA (400 MHz) and an IBM SP2 (66 MHz) (see Ref. 18).

COCOA was found extremely suitable for our numerical simulations. One of the real benefits of inexpensive machines is that they do not have to be shared with hundreds of other users, and we do not have to wait days in a queueing system. We quite often have to wait several hours at a supercomputer center just to use a few processors. In addition, while it is quite difficult to get 50,000 CPU hours at a supercomputer center, the COCOA Beowulf cluster provides more than 400,000 CPU hours per year. Furthermore, processors on parallel supercomputers are usually *at most* twice as fast as these PC processors for large production codes. COCOA was also found to have good scalability with most of the MPI applications used. Although Beowulf clusters have very high latency as compared to conventional supercomputers, this has not been a factor for many important computational fluid dynamics applications. For those codes that have high communication to computation ratios, COCOA was not found a suitable platform because of the high latency. Faster interconnect networks (such as Myrinet) could be used to decrease latency, but these would increase the system cost by about 50%.

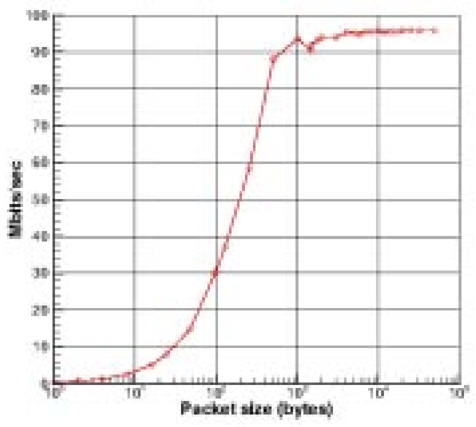

Figure 6. Network timings on COCOA (see Ref. 18).

CONCLUSIONS

We have presented two examples of self-scheduling parallel computer programs that allow one to run numerous copies of a serial algorithm. The first example was a Gaussian elimination problem, representative of aerodynamic panel methods or electromagnetic moment methods. The results showed essentially 100% parallel efficiency, since there was virtually no interprocessor communication.

We also presented a parallel version of the noise prediction code WOPWOP that will permit hundreds or thousands of instances of WOPWOP to be run very easily. We also discussed how the input and output can be organized and presented from all the different runs. The use of the Message Passing Interface (MPI) to run multiple copies of a serial code was found to be a very effective way to obtain results over large parameter spaces.
Since the approach outlined here is applicable to a wide variety of engineering and scientific problems, we have described the approach in

detail, and included a sample code in the Appendix, which can be used as a template. While grand challenge type problems have been run on parallel supercomputers for many years, there have been few engineering applications that have utilized these machines. The approach outlined here will allow more people to use parallel supercomputers and workstation clusters, especially in a design environment where large parameter spaces must be investigated.

REFERENCES

[1] Chatterjee, R., Postawa, Z., Winograd, N., and Garrison, B. J., "Molecular Dynamics Simulation Study of Molecular Ejection Mechanisms: keV, Particle-bombardment of C6H6/Ag {111}," *J. Phys. Chem*. B, 103, 151-163 (1999).
[2] Nwobi, O.C., Long, L. N., and Micci, M. M., "Molecular Dynamics Studies of Thermophysical Properties of Supercritical Ethylene," *J. of Thermophysics and Heat Transfer*, Vol. 13, No. 2, April–June, 1999.
[3] Kaltz, T. L., Long, L. N., and Micci, M. M., "Supercritical Vaporization of Liquid Oxygen Droplets using Molecular Dynamics," *Combustion Science and Technology*, 1998.
[4] Anderson, J. B., Traynor, C. B., and Boghosian, B. M., An Exact Quantum Monte Carlo Calculation of the He-He Interaction Potential, *J. Chem. Phys.* 99, 345, 1993.
[5] Kantor, A., Long, L. N., Micci, M. M., "Molecular Dynamics Simulation of Dissociation Kinetics," AIAA Paper No. 2000-0213, Reno, NV, Jan., 2000.
[6] Strawn, R. C., Oliker, L., and Biswas, R.; "New Computational Methods for the Prediction and Analysis of Helicopter Noise," *J. of Aircraft*, Vol. 34, No. 5, Sept., 1997.
[7] http://www.mhpcc.edu/training/
[8] http://www.personal.psu.edu/lnl/424/
[9] Gropp, W., Lusk, E., Skjellum, A., "Using MPI," MIT Press, 1994.
[10] http://www.personal.psu.edu/lnl/424/mpi/pgauss.f
[11] Brentner, Kenneth S., "Prediction of Helicopter Rotor Discrete Frequency Noise—A Computer Program Incorporating Realistic Motions and Advanced Acoustic Formulation," NASA TM-87721, Oct., 1986.
[12] Ffowcs Williams, J. E., and Hawkings, D. L., "Sound Generated by Turbulence and Surfaces in Arbitrary Motion," *Philosophical Transactions of the Royal Society*, Vol. A264, No. 1151, 1969, pp. 321–342.
[13] Welsh, M., Linux Installation and Getting Started, http://metalab.unc.edu/mdw/LDP/gs/gs.html.
[14] Farassat, F., and Succi, George P., "The Prediction of Helicopter Discrete Frequency Noise," *Vertica*, Vol. 7, No. 4, 1983, pp. 309–320.
[15] Lighthill, M. J., "On Sound Generated Aerodynamically, I: General Theory," *Proceedings of the Royal Society*, Vol. A211, 1952, pp. 564–587.
[16] http://www.personal.psu.edu/lnl/424/mpi/wwmain.f
[17] http://cocoa.aero.psu.edu/
[18] Modi, A., Unsteady Separated Flow Simulations using a Cluster of Workstations, M. S. Thesis, Penn State University, May, 1999.
[19] Modi, A. and Long, L. N., "Unsteady Separated Flow Simulations using a Cluster of Workstations," AIAA Paper 2000-0272, Reno, NV, Jan., 2000.

APPENDIX:

Self-Scheduling F90/MPI Gaussian Elimination Program

```
!*********************************************************************
!                        PGAUSS.F
!  Self-Scheduling Parallel Driver Routine to Solve Many Linear Systems
!
! One of the interesting features of this code is the 'self scheduling'.
! The master processor does nothing but assign work for the other
! processors.  This approach can be used for many of numerical methods.
! It works especially well when the processors do not have the same
! speed (or some processors are given more work), since it will
! automatically give more work to the faster processors.
!
! For the latest version of this program go to:
!       http://www.personal.psu.edu/lnl/424/mpi/pgauss.f
!
!  Prof. Lyle Long, The Pennsylvania State University
!  lnl@psu.edu,   April 16, 1999
!***********************************************************************
      program main
      implicit none
      include 'mpif90.h'

      integer myid, numprocs, ierr, status(MPI_STATUS_SIZE),nn, &
           i,j, numsent, sender, rowtype,anstype,donetype, rows,cols,total
      parameter ( nn = 5000 )
      real*8 ans, ops, tstart,tend, a(nn,nn), b(nn), x(nn)
      character*15 myfile

      ans = 1.0

! Initialize MPI

      call MPI_INIT( ierr )                               !   Initialize MPI
      call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )    !   Find myid
      call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )!   Find numprocs

      print *, "Process ", myid, " of ", numprocs, " is alive"

      if (numprocs .lt. 2) then
         print *, "Master-Slave code must have at least 2 processes!"
         call MPI_ABORT( MPI_COMM_WORLD, 1 )
         stop
      end if

      total = 120   ! This sets how many linear systems will be solved

      rowtype  = 1
      anstype  = 2
      donetype = 3

      tstart = mpi_wtime()
      if ( myid .eq. 0 ) then  !----This is the Master processor Code---
        print *, ' Number of Processors                  = ', numprocs
        print *, ' Size of Matrices                      = ', nn, ' by ', nn
        print *, ' Total number of Linear Systems Solved = ', total
        numsent = 0

        do i = 1, total

           numsent = numsent + 1

!    Wait for a message from any processor, before sending them work:
           call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,&
                         anstype, MPI_COMM_WORLD, status, ierr)

!    this tells you which processor is ready for work:
           sender = status(MPI_SOURCE)

!    send more work to the same processor that just finished:
           call MPI_SEND(numsent, 1, MPI_INTEGER, sender,&
                         rowtype, MPI_COMM_WORLD, ierr)

           print *, ' Processor ', sender, ', working on matrix no. ',i

        end do

!    When done, tell all the processors to stop

        do i = 1, numprocs-1
           call MPI_RECV(ans, 1, MPI_DOUBLE_PRECISION, MPI_ANY_SOURCE,&
                         anstype, MPI_COMM_WORLD, status, ierr)
           sender = status(MPI_SOURCE)
           call MPI_SEND( numsent, 1, MPI_INTEGER, sender,&
                          donetype, MPI_COMM_WORLD, ierr)
        end do

      else   ! ----this is the start of the slave processes-------------

 ! Tell the Master you are ready for work:
 90      call MPI_SEND(ans, 1, MPI_DOUBLE_PRECISION, 0, anstype, &
                       MPI_COMM_WORLD, ierr)

 !   receive info from master :
         call MPI_RECV(NUMSENT, 1, MPI_INTEGER, 0,&
                       MPI_ANY_TAG, MPI_COMM_WORLD, status, ierr)

 !   if done quit, otherwise create matrix and solve it:
         if (status(MPI_TAG) .ne. donetype)  THEN

            ! create a random diagonally dominant matrix & RHS
            call random_number(a)
            call random_number(b)
            do j=1,nn
               a(j,j) = 10.0*nn
               b(j)   = 10.0*nn
            end do

            call gauss( a, b, nn, x)    ! solve system of equations

 !   the following writes character info to the character myfile


            write(myfile,900) 'out', NUMSENT, '.dat'
 900        format(a,i3.3,a)
```

```
            open ( 77, file=myfile )
              do i=1,nn
                write(77,*) i, x(i)
              end do
            close( 77 )

            GOTO 90

         END IF   ! end IF for donetype

      end if   ! end master-slave IF-THEN-ELSE construct

      tend = mpi_wtime()

!  have each processor print out it’s cpu time:
      print *, ’ node=’ , myid,  ’, time = ’,tend-tstart,’ seconds’

      if ( myid .eq. 0 ) then
         print *, ’ total megaflops = ’, &
                   1.0e-6 * (2.0/3.0) * nn**3 * total / (tend-tstart)
      end if

      call MPI_FINALIZE(ierr)
      stop
      end
!-------------------------------------------------------------
       subroutine gauss(a,b,n,x)
!-------------------------------------------------------------
!  this routine performs Gaussian elimination.
!  The matrix is ’a’, the right-hand-side is ’b’,
!  and the solution is ’x’.  It does not pivot.
!-------------------------------------------------------------
      implicit none
      integer n, i, j, row
      real*8 a(n,n), x(n), ratio, b(n)

      do i = 2, n    ! forward elimination, loop thru elements 2 thru n
        do row = i, n       ! scale all the rows and add to row i-1
           ratio  = a(row,i-1) / a(i-1,i-1)
           b(row) = b(row)  -   ratio * b(i-1)

           do j = i, n                ! loop over all columns from i to n:
              a(row,j) = a(row,j)  -   ratio * a(i-1,j)
           end do
        end do

      end do                          ! this ends the forward elimination

      x(n) = b(n) / a(n,n)            ! back substitute, first find x(n)

      do i = n-1, 1, -1
        x(i) = b(i) / a(i,i)
        do j = i+1, n
          x(i)   = x(i) - a(i,j) * x(j) / a(i,i)
        end do
      end do

      return
      end
```