PARALLELIZING SIMULATIONS OF NONNEUTRAL PLASMA INSTABILITIES IN A MALMBERG-PENNING TRAP

by

Melissa Powell

A 492R capstone project report submitted to the faculty of

Brigham Young University

in partial fulfillment of the requirements for the degree of

Bachelor of Science

Advisor: Ross Spencer

Department of Physics and Astronomy

Brigham Young University

December 2007

ABSTRACT

PARALLELIZING SIMULATIONS OF NONNEUTRAL PLASMA INSTABILITIES IN A MALMBERG-PENNING TRAP

Melissa Powell

Department of Physics and Astronomy

Bachelor of Science

A Malmberg-Penning trap is a cylindrical apparatus which confines non-neutral plasma with an axial magnetic field and negative electric potentials on both ends. Theory predicts that a hollow plasma density profile is unstable, and experiments agree. However, the experimental growth rate of the m=1 diocotron mode of the instability is much larger than the theoretical growth rate, by a factor of around 2-4. We are collaborating with an experimental research group to find the cause for this discrepancy by recreating their Malmberg-Penning trap in our 3D PIC computer simulation. The growth rates of our simulation test cases have remained roughly half that of the experiments. I will report how we successfully parallelized the simulation, allowing the number of plasma particles to be increased to approximately the number of particles in the experiment. The increased number of particles improves the accuracy of the simulation by reducing the noise on the computational grid.

ACKNOWLEDGMENTS

I gratefully acknowledge the patience and assistance of my research advisor, Dr. Spencer, in completing this project, as well as Dr. Mason and Dr. Campbell's help and support. I also greatly appreciate having the use of the facilities of the Brigham Young University Supercomputing Center, without which this project would not have been possible. This project was supported and funded by NSF and the Brigham Young University Physics Department.

Contents

Chapter 1

Introduction

The behavior of plasma, or in other words gases of charged particles, is a subject of interest in many fields, including technology and astronomy. The waves that can be created in pure electron plasma confined in an apparatus such as the cylindrically shaped Malmberg Penning trap are of special interest because they can be used to check the theory of fluid dynamics. Much of the theory of plasma physics and fluid dynamics has been verified by studying the waves in confined electron plasma.

The diocotron instability waves created in electron plasma confined in a cylindrical trap when the column of electron gas has a hollow radial density profile were first given serious theoretical consideration by Levy [1] in 1965. He predicted that the m=1 mode of the diocotron instability should be stable, while m $\geq$ 2 modes should be unstable. The application of fluid dynamics theory to pure electron plasma confined in a cylindrical apparatus has since been used to successfully predict much of the diocotron wave behavior in experiments [2]. However, in 1990 Driscoll and Fine [3,4] reported that they had observed an exponentially growing m=1 mode of the diocotron instability, contrary to the theory which said that the m=1 mode should be stable. Soon afterward, Smith [5] reported that their investigations using a particle-in-cell simulation showed how m=1 exponential growth could be created, but they had a lower growth rate than that observed in the experiments. Many attempts were made to theoretically explain the high growth rate of the m=1 mode [6, 7, 9]. However, no satisfactory explanation has yet been found for the discrepancy between theory and experiment in the growth rate of the m=1 mode.

Our research group seeks to understand the cause of the discrepancy by duplicating experimental conditions as closely as possible in computer simulations, collaborating with Travis Mitchell's experimental research team at the University of Delaware. Our simulation of non-neutral plasma is discussed in a paper by Spencer and Mason [7]. The simulation calculates the motion of the plasma confined in a Malmberg-Penning trap using the equations of the Drift-Poisson Model, and runs on a 3D computational grid in Cartesian coordinates.

Several simulations have been created to explore the behavior of non-neutral plasma confined in a cylindrical trap. An excellent overview of these simulations is given by Tsidulko et al [8], who noted that while many 2D particle-in-cell codes have been created, they knew of only a few which are 3D. For example, the simulation by Smith [5] that was cited above was 2D and had a total of 32,000 particles. The simulation recently developed by Tsidulko et al was 3D and was run on a single processor on a personal computer [8]. Although many parallelized 3D PIC simulations have been developed for many areas of plasma and fluid study, according to our knowledge no other simulation has been developed for this particular research area besides our own which is both 3D and parallelized to allow access to more memory and hence more particles than would be possible with a single processor. In the research which I will describe in this paper, we have parallelized our simulation and increased the number of particles from 20 million to 2 billion, which is on the same order of magnitude as the number of electrons in the experiment. This was done by running the simulation with 100 processors on the supercomputer marylou4 at the BYU Supercomputing Center, which has a total of 1,260 processors and 5,040 GB of total memory. The memory made available by using 100 processors allowed us to run with 100 times as many particles as before. The increased number of particles results in less noise and better statistics on the computational grid.

Chapter 2

Methods

2.1 The Malmberg-Penning Trap

We simulate the Malmberg-Penning trap used by Travis Mitchell's experimental research team in our particle-in-cell code. Malmberg-Penning traps were first described in 1975 by Malmberg and deGrassie [10] and are an effective means of confining and studying waves in electron plasma. This trap consists of a cylindrical apparatus which confines a column of pure electron plasma with an axial magnetic field and split rings on both ends holding negative potentials of -50 V. Fig. 2.1 is a rough sketch of the trap showing the electric and magnetic fields inside the cylinder, while Fig. 2.2 is a more detailed diagram showing diagnostic rings and wave control rings that can be used to manipulate the plasma [11]. A plot of the electron density at a given moment in time is obtained by dumping the entire plasma column out one end onto a phosphor screen.

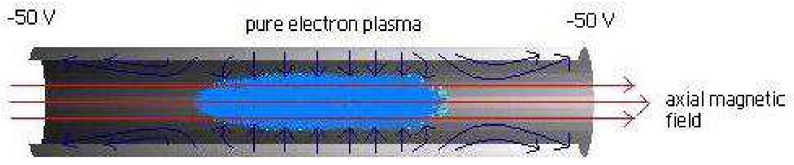


Figure 2.1 A sketch of the electric(blue) and magnetic(red) fields inside the cylindrical apparatus confining the plasma in a MP trap. Plasma electrons spin around the magnetic field and so cannot escape radially.

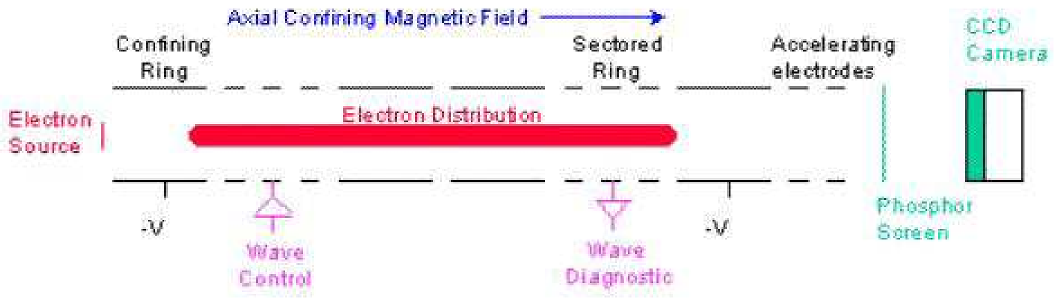


Figure 2.2 A diagram of an MP trap showing diagnostics and controls. Electron density is measured at a given point in time by dumping the entire electron column out one end onto the phosphor screen, yielding 2D plots of electron density. Fig. from [11].

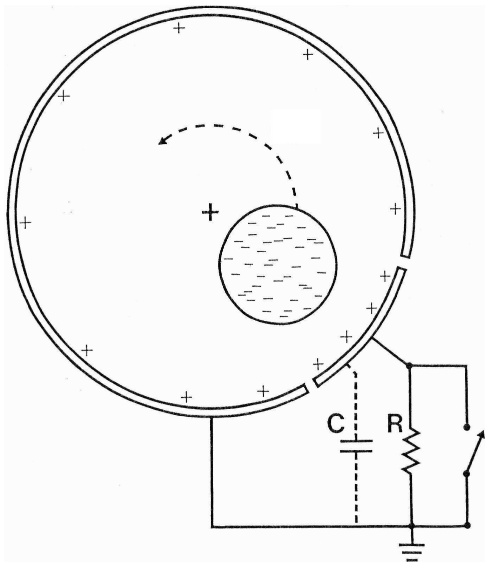


Figure 2.3 The electron column precesses around the conducting cylinder when it is in a stable state. Fig. from [2].

2.1.1 Diocotron Instabilities

In its stable state, the electron column has high density near the center and lower density near its edges, and precesses about the center as in Fig. 2.3. This stable mode is due to the induced charge on the walls and the axial magnetic field. In order to set up instabilities within the plasma, the column is hollowed by evacuating electrons from the middle. Non-neutral plasma has fluidlike properties, and in fact is used to study the behavior of fluids because motion in fluids is in some ways more difficult to study than in electrons [11]. To picture what happens when the plasma is hollowed, imagine if a bucket of water were suddenly taken from a rainbarrel. Where before the water was calm, now it is unstable and turbulent.

Similarly, when the plasma column is hollowed, the plasma becomes turbulent. The entire column no longer moves together as it precesses around the center. Instead, a part of the plasma precesses in one direction and part of it in another, and instabilities called diocotron waves are created [2]. In particular, the m=1 instability mode amplitude grows until the plasma reforms and becomes stable again with the highest density in the center. The electron density plots in Fig. 2.4 show this process for the m=2 instability. The m=1 instability causes a similar evolution. Experiments with these diocotron instabilities are described in detail in papers by Driscoll [3], Malmberg [2], and Fine, Driscoll and Malmberg [12].

2.2 Overview of the Simulation

Since experiment and theory do not agree on the exponential growth rate of the m=1 instability, our simulation attempts to bridge the two by simulating the experiment using plasma theory and adjusting various parameters such as the potentials on the rings in order to try to recreate the experiment's high m=1 growth rate.

The positions of the plasma particles within the cylinder are stored in Cartesian coordinates. Plasma particle motion is calculated using the equations of the Drift-Poisson Model, which are as follows.

Newton's Second Law for motion in the z-direction (axial direction) is

$$m\frac{dv_z}{dt} = -q\frac{\partial\phi}{\partial z} \quad , \quad \frac{dz}{dt} = v_z(t) \quad , \tag{2.1}$$

where $\phi = \phi(x, y, z)$ is the electrostatic potential.

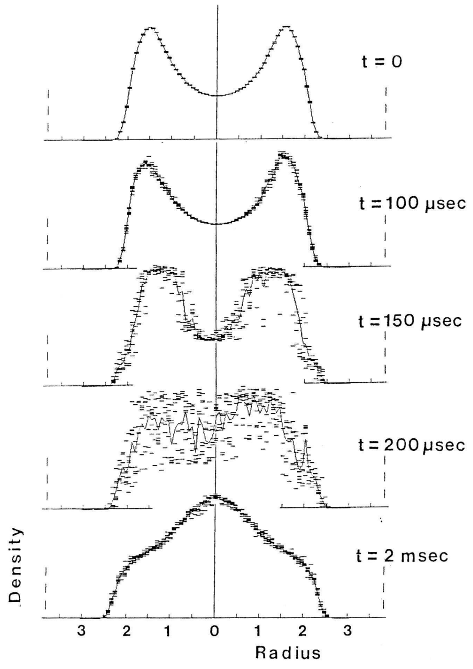
In the xy-plane motion is governed by the electric field drift:

$$\frac{d\mathbf{r}_\perp}{dt} = \frac{\mathbf{E}\times\mathbf{z}}{B} \tag{2.2}$$

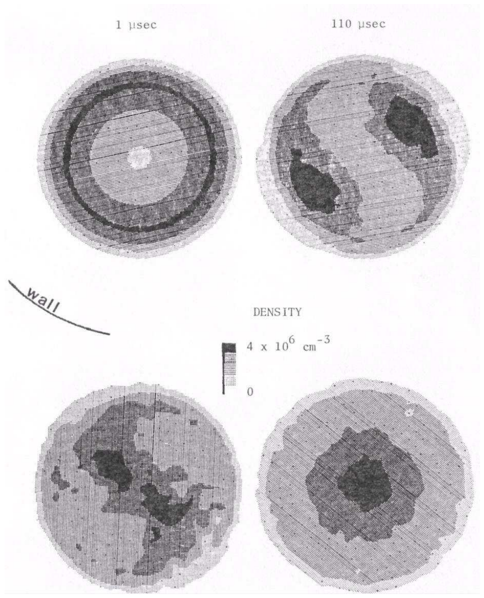
where $\mathbf{r}_\perp = x(t)\mathbf{x} + y(t)\mathbf{y}$ and where $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{z}$ are the three cartesian unit vectors.

The electrostatic potential is given by Poisson's equation:

$$\nabla^2\phi = -\frac{q}{\epsilon_0} \quad \mathrm{n} \tag{2.3}$$



(a)



(b)

Figure 2.4 These plots are snapshots in time of the electron density profile across the plasma column's radius in experiments described by Malmberg [2], and show the evolution of the plasma density profile caused by the m=2 instability. The m=1 instability causes a similar evolution. a) At $t = 0$, the plasma column has just been hollowed by evacuating electrons from the middle of the column (near 0 radius). Notice how the density graph has become scattered by $t = 200$ μsec by the increasing instabilities in the plasma caused by this unstable shape. At $t = 2$ msec, the instabilities have died off and the plasma has reformed itself into a stable state (a non-hollow shape). b) After the plasma column is hollowed, part of the plasma precesses in one direction and part of it in another and instability modes grow rapidly. Fig. from [2].

with

$$\mathbf{E} = -\nabla\phi \tag{2.4}$$

where n = n(x, y, z) is the plasma number density. The guiding center approximation [13] is used so that a particle's motion follows the path of the time-averaged center of mass of an electron.

The simulation code runs in a three step cycle as seen in Fig. 2.5. A time step of 3×10^{-9} sec is completed each cycle. The three-step cycle to move the particles is as follows:

1. ***Resolve Particles into Density Grid*** Each particle contributes a little of its charge density to each of the eight points of the grid cube it is in to compute a 3D charge density grid.
2. ***Field Solve*** The forces acting on the particles are found by using the charge density grid and multigrid numerical techniques to solve for the electric potential in Poisson's equation.
3. ***Move Particles*** The particles are moved using the forces found by the field solver.

The simulation is typically run for between 3000 and 30000 time steps. The motion of the plasma is analyzed by graphing file output diagnostics such as the amplitude of the m=1 mode (as was done in Fig.3.1).

2.3 Parallelizing the Simulation

2.3.1 Using the MPI Library

The Message-Passing Interface(MPI) library provides an orderly way for communication to take place between processors executing code [14]. We used MPI to parallelize our code because its subroutines provide good control over processor communication and the library is widely used. MPI Fortran and C/C++ compilers are included

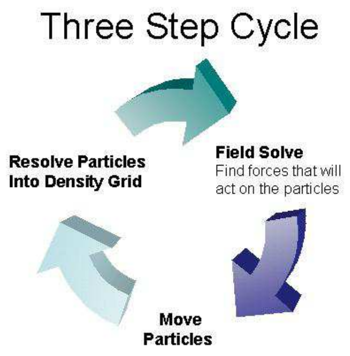
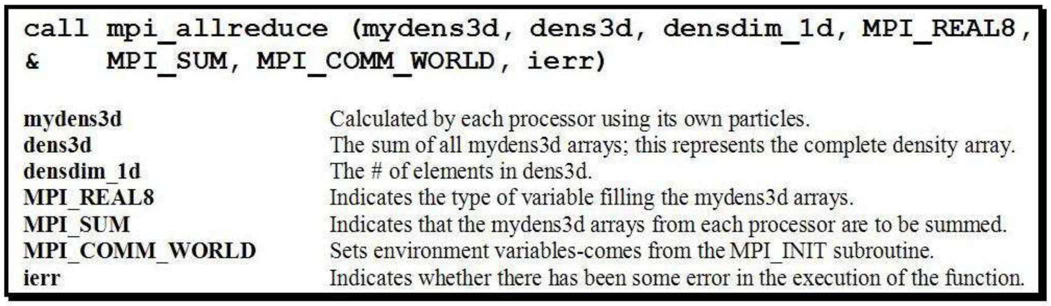

Three Step Cycle
Field Solve
Find forces that will act on the particles
Move Particles
Resolve Particles Into Density Grid

Figure 2.5 A time step in the simulation represents the completion of one cycle.

on the marylou4 supercomputer in the BYU Supercomputing Laboratory. A sample MPI Fortran program is included in Appendix A, along with information on how to submit parallel processing jobs to run on marylou4.

The MPI subroutines allow direct control over communication between processors with the subroutines. For instance, the MPI subroutine send and receive allows the programmer to command one processor to send a packet of data(like an array) and another processor to receive it and store it. A more complicated subroutine called mpi_reduce will collect the indicated arrays stored by the same name but holding different values on all processors and perform some operation on the arrays, such as add or subtract. A very useful guide on MPI Fortran subroutines can be found at the Texas Advanced Computing Center's website [15]. The control over processor communication provided by MPI subroutines makes it easy to understand and modify a code's algorithm. Careful consideration of our code's algorithm was necessary in order to keep communication time overhead from canceling out the benefits of using multiple processors.



```
call mpi_allreduce (mydens3d, dens3d, densdim_1d, MPI_REAL8,
&    MPI_SUM, MPI_COMM_WORLD, ierr)
```

mydens3d	Calculated by each processor using its own particles.
dens3d	The sum of all mydens3d arrays; this represents the complete density array.
densdim_1d	The # of elements in dens3d.
MPI_REAL8	Indicates the type of variable filling the mydens3d arrays.
MPI_SUM	Indicates that the mydens3d arrays from each processor are to be summed.
MPI_COMM_WORLD	Sets environment variables-comes from the MPI_INIT subroutine.
ierr	Indicates whether there has been some error in the execution of the function.

Figure 2.6 The MPI subroutine mpi_allreduce collects the mydens3d arrays from all processors, adds them together, and stores the result in dens3d.

2.3.2 Parallelizing the Densmaker and Mover Steps

In the regular serial version of our code, the 3D density array called dens3D stores the charge density grid as calculated in the densmaker function in step 1 in the three step cycle in Section 1.3. Dens3d is used by the field solver to calculate the fields, which are then used to calculate the particle moves.

In the parallelized version, each processor stores about 20 million particles in its particle position arrays, which is close to the maximum the compiler will allow given the available memory. In the densmaker function, each processor calculates its own dens3d array for its own particles, called mydens3d. Thus the resolving density step is parallelized, since the work to create the complete density array is divided up among processors working in parallel. When all processors have created their mydens3d arrays, mpi_allreduce is called to collect the arrays, add them together, and send the result to all processors to be stored as their copy of dens3d. The full call to mpi_allreduce is shown in Fig. 2.6.

In the next step, the field solve, each processor uses its own copy of the dens3d array to calculate the fields. Since they are all doing the same thing here, there is some wasted processor time, and the field solver step is not truly parallelized. Once the field solver step is complete, each processor then moves its own set of particles, so that the mover step as well as the densmaker step is parallelized. In summary, we successfully parallelized the simulation to the extent that we can increase the number of particles in proportion to the number of processors that we run our simulation on.

Chapter 3

Results

Parallelizing the simulation allowed us to increase the number of particles in the simulation from 20 million to 2 billion, which is on the same order of magnitude as the actual number of electrons in the experiment. This was made possible by running the simulation with the memory made available by 100 processors on marylou4. The number of particles per processor was set to 20 million since that was close to the maximum the compiler would allow given the available memory.

The increased number of particles results in less noise and better accuracy in the simulation. For example, previously we were forced to have a much higher starting amplitude than in the experiment because the high noise levels with the number of particles we had in the simulation made observing low amplitudes impossible. In the experiment, the m=1 component of the perturbed density, normalized to the maximum equilibrium density, starts at about 2×10^{-4} and increases exponentially to about 5×10^{-3}, after which it levels off. Our simulation's relative amplitude started much higher, at about 1×10^{-2}, which we hypothesized was making it difficult to reproduce the experimental conditions.

We found that running with 100 processors and thereby increasing the number

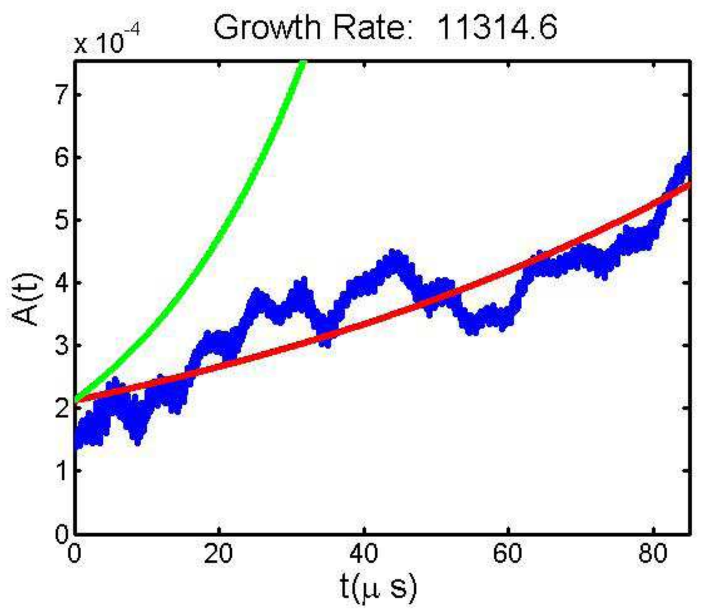


Figure 3.1 A run using a total of 2 billion particles. The green curve shows a 4×10^4 growth rate for comparison.

of particles from 20 million to 2 billion decreased the noise level sufficiently to allow us to reach a starting relative amplitude close to the experimental value. The m=1 amplitude plot of such a run is seen in Fig. 3.1. In a run using 100 processors with a total of 2 billion particles, we were able to achieve a starting relative amplitude close to the experimental value of 2×10^{-4}. We hoped that this improvement in the accuracy of our simualtion would result in a growth rate more like that in the experiment's. However, the growth rate of 1.13×10^4 was still much lower than in the experiment, which had growth rates from 4×10^4 to 8×10^4. We would like to further improve the accuracy of the simulation by making the field solver more efficient by parallelizing it and converting it from a two grid to a three grid multigrid solver. If the field solver could do more work in less time, we could use a finer grid, which would be more like the physical system. For comparison, the run in Fig. 3.1 completed 29,000 time steps in a period of 10 days. If we had used a significantly finer grid it would have taken much longer to complete the same number of time steps.

Chapter 4

Conclusion

In conclusion, parallelizing our simulation has allowed us to significantly increase the number of particles and reduce the noise on the grid. Using 100 processors, we were able to have a maximum of 2 billion particles, a number that approaches the actual number of electrons in the experiment. Although getting the starting amplitude closer to the experimental value did not turn out to have a significant effect on the growth rate during the 10 day run seen in Fig. 3.1, we may be able to further improve the accuracy of the simulation by running with a finer grid. However, a finer grid is more computationally expensive and therefore significantly increases the runtime. We have been working on increasing the efficiency of the field solver by modifying its multigrid algorithm to include three grids instead of two. Adding another, coarser grid in the V-loop of the multigrid algorithm should make the long wavelength error disappear faster. We may also be able to split up the work during the field solver step and take advantage of the increased number of processors, although the interdependency of the steps of the multigrid algorithm complicate its parallelization. Thus, in the future parallelizing and optimizing the simulation's field solver should further improve the accuracy of the simulation by allowing runs on a finer grid.

Bibliography

[1] R.H. Levy, ”Diocotron Instability in a Cylindrical Geometry,” Phys. Fluids 8, 1288 (1965).

[2] J.H. Malmberg et al, “Experiments With Pure Electron Plasmas,” AIP Conference Proceedings 175, 28 (1988).

[3] C.F. Driscoll, “Observation of an Unstable l=1 Diocotron Mode on a Hollow Electron Column,” Phys. Rev. Lett. 64, 645 (1990).

[4] C.F. Driscoll, K.S. Fine, ”Observation of an Unstable l=1 Diocotron Mode on a Hollow Electron Column,” Phys. Fluids B 2, 1359 (1990).

[5] R.A. Smith, M.N. Rosenbluth, ”Algebraic instability of hollow electron columns and cylindrical vortices,” Phys. Rev. Letters, 64, 649 (1990).

[6] T.J. Hilsabeck, T.M. O’Neil, ”Finite Length Diocotron Mode,” Phys. Plasmas 8, 407 (2001).

[7] G.W. Mason, R.L. Spencer, ”Simulations of the instability of the m = 1 self-shielding diocotron mode in finite-length nonneutral plasmas,” Phys. Plasmas 9, 3217 (2002).

[8] Y. Tsidulko, R. Pozzoli, M. Romé, "MEP: A 3D PIC code for the simulation of the dynamics of a non-neutral plasma," Journal of Computational Physics 209(2), 406 (2005).

[9] S.N. Rasband, R.L. Spencer, "Modes in a non-neutral plasma of finite length m=0,1," Phys. Plasmas 10, 948 (2003).

[10] J.H. Malmberg, J.S. deGrassie, "Properties of Nonneutral Plasma," Phys. Rev. Letters 35, 577 (1975).

[11] T.B. Mitchell, "Travis Mitchell's Research Interests: Penning Traps, 2D Electron Fluids," http://www.physics.udel.edu/wwwusers/mitchell/research_interests.html (Accessed June 22, 2007).

[12] K.S. Fine, C.F. Driscoll and J.H. Malmberg, "Measurements of a Nonlinear Diocotron Mode in Pure Electron Plasmas," Phys. Rev. Lett. 63, 2232 (1989).

[13] T.J.M. Boyd, J.J. Sanderson, The Physics of Plasmas, Cambridge University Press, 2003, pp. 17-19.

[14] W. Gropp, E. Lusk, A. Skjellum, Using MPI: Parallel Programming with the Message-Passing Interface, MIT Press, 1994, pp. 11-23.

[15] Texas Advanced Computing Center, "MPI for Fortran Programmers," http://www.tacc.utexas.edu/resources/user_guides/mpi/ (Accessed June 22, 2007).

Appendix A

Using Multiple Processors on Marylou4

Using Multiple Processors on Marylou4

Melissa Powell

This is some useful information on how to get started on running MPI Fortran code with multiple processors on marylou4 at the BYU Supercomputing Center, after getting an account that has permission to run on the supercomputers.

Basic info on marylou4

Ira and Mary Lou Fulton Supercomputing Center http://marylou.byu.edu/
marylou4 Dell 1955 Blade Cluster
1260 Dual Core Intel EM64T processors @ 2.6 GHz
618 compute nodes
5,040 GB total memory (8 GB/node)
15 TB Disk
RedHat Enterprise Linux v4.3
PBS submission system
MPI-enabled Fortran 77/90, C/C++ compilers

.tcshrc

This is a sample .tcshrc file, adapted from Grant W. Mason's .tcshrc file. This file should be at the top most level of your account directory. As long as it is present, it will be used to set up your account preferences and set your path variable, which is all the directories the shell will search when you type commands. Note the **/opt/mpich/intel/bin/mpif90** which will allow you to compile with the MPI Fortran 90 compiler.

```
set prompt='[%m:%c] %n> '

setenv TERM vt220

set path = (. ..              \
                    $path                        \
                    ~/bin                        \
                    /usr/gm                      \
                    /usr/bin                     \
                    /usr/local/bin               \
                    /bin                         \
                    /usr/X11R6/bin               \
                    /opt/mpich/intel/bin/mpif90  \
                    )

alias ll ls -la

limit stacksize unlimited
limit coredumpsize 0
```

Core dump is data left over after a program terminates abnormally. If you don't want to look at these leftovers, set the coredumpsize limit to 0.

Compile on marylou4 with: /opt/mpich/intel/bin/mpif90 -o MyExe MyMPIProg.f
Our simulation was compiled with the following options(*italicized*):
/opt/mpich/intel/bin/mpif90 -o MyExe MyMPIProg.f *-cm -save -O3 –xP*

SumProg.f

This is an example of MPI Fortran code. The MPI(Message Passing Interface) library can be used to tell processors what to do and how to communicate during runtime.

```
c         SumProg
c         Melissa Powell
c         Demonstrates important aspects of an MPI program
c         This program finds the definite integral of a function, dividing up the task
c         according to the number of processors assigned to it in the PBS script.

          program SumProg
          implicit none
c         To include MPI functions,
          include 'mpif.h'

c         MPI variables
          integer :: myid, numprocs,ierr

c         Functions
          real*8 :: y_value

c         Program variables
          real*8 :: i,j, dx, height
          integer :: standard_myN, myN
          double precision :: sum
          real*8 :: mysum = 0
          real*8 :: a = 1
          real*8 :: b = 2
c         High N is just to make everything take longer(not to get high accuracy)
c         so you can see how runtimes vary for different numbers of processors
          integer :: N = 1000000000

c         You must call MPI_INIT if you want to use multiple processors
          call MPI_INIT(ierr)

c         These functions set the values of useful variables you can use
c         to command different processors to do the tasks you want
          call MPI_COMM_SIZE(MPI_COMM_WORLD, numprocs, ierr)
          call MPI_COMM_RANK(MPI_COMM_WORLD, myid, ierr)

          if (myid .eq. 0) then
                    write(*,*) 'STARTING PROGRAM- SumProg'
                    write(*,*) 'Number of processors: ', numprocs
          end if

c         write(*,*) '  This processor is #', myid
```

```
c         Setting limits of integration for
c         each processor:

c         myN for all processors but the last
          standard_myN = int(N/numprocs)

c         If I am not the last processor, then
          if (myid .lt. numprocs-1) then
                    myN = standard_myN*(myid+1)
          else
                    myN = N
          end if
          write(*,*) 'myid, myN: ', myid, myN

c         Summing area under the curve between this
c         processor's set limits of integration:

c         Do loop is just to make everything take longer(for time tests)
          dx = (b-a)/N
c         do j = 1, 1000
                    mysum = 0
                    do i = standard_myN*myid, myN-1
                    height = y_value(a + .5*dx + real(i)*dx)
                    mysum = mysum + height*dx
                    end do
c         end do
          write(*,*) 'My part of the Def Int: (myid, sum)', myid, mysum

c         This MPI function sums the results each processor got
c         for the area between its limits of integration:
     call MPI_REDUCE(mysum, sum, 1, MPI_DOUBLE_PRECISION,
    &            MPI_SUM, 0, MPI_COMM_WORLD, ierr)

c         You must call MPI_FINALIZE when you're done
c         or strange EOF errors will occur
          call MPI_FINALIZE(ierr)

          if (myid .eq. 0) then
                    write(*,*) 'Definite integral: (myid, sum)', myid, sum
                    write(*,*) 'ENDING PROGRAM'
          end if
          stop
          end program SumProg

          real function y_value(num)
          implicit none
          real*8 :: num
          y_value = 3*num**2
          return
          end
```

Running with Multiple Processors(PBS)

Programs can be run with multiple processors by submitting them to the queue with PBS. PBS assigns the number of processors requested to the jobs in the job queue so that they can start running. The supercomputing center's system managers use PBS to keep track of the jobs being run to make sure nobody is misusing the resources.

- -The walltime set in the PBS submission file keeps your program runtime within a reasonable limit.
- -Nodes are groups of processors, about 1-4 each on marylou4.
  - -Running within nodes makes your program run faster.

SumProg.sh

A sample PBS submission script file for SumProg adapted from http://marylou.byu.edu/pbs.php, where there is a complete tutorial on the script commands.
The submission script sets your preferences for the job when PBS runs it, including how many processors you want and whether you want email notification when the job is done.
Note: The higher you set the walltime, the longer it will take for your job to start running. A recent change in policy does not allow more than 128 processor weeks per user, or more than 480 processors per faculty member at a time.
Ex. You can set NP to 64 and walltime to 2 weeks, or set NP to 1 processor and walltime to 128 weeks. Processor time is NP * walltime.

```
#!/bin/bash
#PBS -l nodes=2:ppn=2,walltime=72:00:00
#PBS -N SumProg

#PBS -m abe
#PBS -M myemail@byu.edu

PROG=/ibrix/home/username/myprogramsdir/SumExe
PROGARGS=""

TMPDIR=/ibrix/home/username/tempdir/$PBS_JOBID

# NP should always be nodes * ppn from the #PBS -l directives above
NP=4

#cd into the directory where I typed qsub
cd $PBS_O_WORKDIR

# Execute the mpi job. 'time' tells you how fast it runs.
time mpiexec $PROG $PROGARGS

exit 0
```

Working with the Queue on the PBS system on Marylou4

Submit Job to the Queue **qsub** + *submission file*

```
username> qsub SumProg.sh
162007.m4bi                        *Note: 162007 is your job's ID #
```

Look at the Queue **showq**

```
username> showq
active jobs------------------------
JOBID    USERNAME   STATE    PROC   REMAINING      STARTTIME
945570     wesb      Running   1     57:01:15:18   Wed Oct 11  21:18:34
945522     glh43     Running   1     53:05:49:03   Mon Sep 25 09:50:22
Etc.
```

Delete Job from the Queue **qdel** + *Job ID#*

```
username> qdel 162007
```

Program Output Files

After your program is executed, output files will be put in the same directory qsub was typed in. If you want to pipe output to an output file during runtime so you can watch the program's progress, include something like '> mylogfile.fil' in the PROGARGS in the .sh job submission script. Otherwise the output will be put in the .o output file as below (the .o, .e files are not created until after the job is complete).

SumProg.o162007

Sample output file for SumProg with 4 processors

```
STARTING PROGRAM- SumProg
 Number of processors:           4
 myid, myN:            0   250000000
 myid, myN:            2   750000000
 myid, myN:            1   500000000
 myid, myN:            3  1000000000
 My part of the Def Int: (myid, sum)          3   2.64062500000949
 My part of the Def Int: (myid, sum)          0  0.953124999995298
 My part of the Def Int: (myid, sum)          1   1.42187499999796
 My part of the Def Int: (myid, sum)          2   1.98437500003173
 Definite integral: (myid, sum)          0   7.00000000003447
 ENDING PROGRAM
```

SumProg.e162007

Sample error output file with output from 'time' command in PBS submission script.

```
real    0m9.869s
user    0m1.345s
sys     0m0.087s
```