Assignment 5
Suzaku Programming Assignment

B. Wilkinson: Modification date March 11, 2016. **Correction March 26, 2016 p. 11/12**

Overview

This assignment explores using Suzaku routines to create MPI programs. Suzaku is a pattern parallel programming framework developed at UNC-Charlotte that enables programmers to create pattern-based MPI programs without writing MPI message passing code implicit in the patterns. The purpose of this framework is to simplify message passing programming and create better structured and scalable programs based upon established parallel design patterns. Suzaku is implemented in C and provides both low-level message passing patterns such as point-to-point message passing and higher-level patterns such as the workpool pattern. Several unique patterns and features have been developed including a generalized graph pattern that enables any pattern that can be described by a directed graph to be implemented and a dynamic workpool for solving application such as the shortest path problem.

Suzaku draws from the Seeds framework and from OpenMP. The structure is OpenMP-like but creates distributed-memory MPI code rather than shared-memory thread-based code. In Suzaku, sections of code can be executed by a single process (the master process), which corresponds to the default situation in OpenMP for the main thread before any directives. Parallel sections of code can be created that are executed by all the processes. This corresponding directive in OpenMP is the parallel directive. Within a Suzaku parallel section, various patterns can be used, including broadcast, scatter, gather, master-slave, and higher level patterns such as a workpool.

There are many ways one could define Suzaku routines. *The simple Suzaku macros described here are not defined in the same way as the version 0 Suzaku macros/routines described in the Fall 2014 class.* The idea now is hide MPI completely and have patterns pre-implemented. You will be asked your opinions and ways one could improve Suzaku. *As a new tool, please watch for announcements.*

Part 1 provides basic practice in coding, compiling, and running Suzaku programs. All the programs are given and the applications are familiar. One simply sends messages and data for one process to another process. The second is a matrix multiplication program using a master-slave pattern. The third uses the workpool pattern to implement the Monte-Carlo π calculation.

Part 2 asks you to write a sequential program for the astronomical *N*-body problem, beginning with a sequential program. A template for the sequential program is given. You are then asked to add X11 graphical output.

Part 3 asks convert the sequential *N*-body program into an MPI program using Suzaku routines using a master-slave pattern. This involves mostly deleting one for loop in the sequential program that is not needed when the program is parallelized and adding Suzaku routines at various places to broadcast data and collect results.

Part 4 is for extra credit and asks you to reformulate the *N*-body program as a Suzaku workpool. In a workpool, the number of slaves does not need to be the same as the number of bodies.

Preliminaries

This assignment will be done on your computer only. We are not too concerned with execution speed. We are more concerned about simplifying parallel programming with higher-level tools. The programs you will need are:

- **suzaku.h**
- **suzaku.c**
- **SZ_pt-to-pt-c**
- **SZ_matrixmult.c**
- **MontePi_workpool.c**
- **Matrixmult_workpool,c**
- **makefile** (optional but useful)

which can be found on the course VM in the directory **ParallelProg/Suzaku**. If you are using your own Linux installation, the files can be found on the course home at: "Parallel Programming Software" under "Suzaku." Download all the files and place them in the directory **ParallelProg/Suzaku**.

Significant changes have been made to the Spring 2015 version of Suzaku, so make sure you are using the most recent files as posted, not any files from previous semesters.
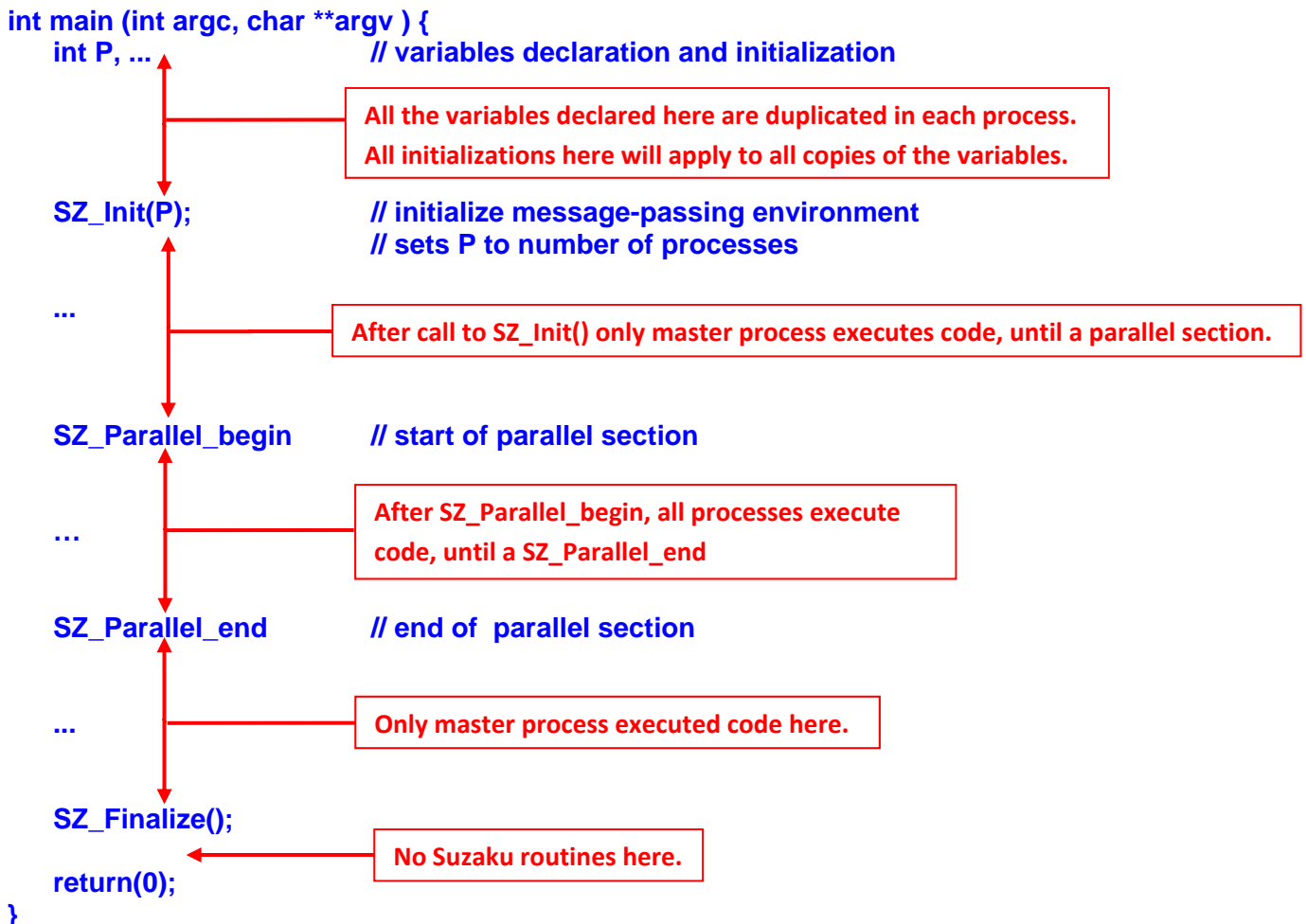
You will also need to refer to posted document "Suzaku Pattern Programming Framework Specification" on the course home at "Additional Information" under "Suzaku" for full details on the signatures of the Suzaku routines, their usage, and limitations.

Part 1 – Suzaku Tutorial (35%)

The purpose of this part is to become familiar with Suzaku so that you will be able to use them in your own programs in subsequent parts. All the programs are given and you are simply asked to compile and execute them. This part should not take long (10 minutes!). However carefully review the provided programs so that you fully understand them.

Program Structure

The computational model is similar to OpenMP but using processes instead of threads. With the process-based model, there is no implicit shared memory. The structure of a Suzaku program is shown below. The computation begins with a single master process (after declaring variables that are duplicated in all processes and the initialization of the environment). One or more parallel sections can be created that will use all the processes including the master process. Outside parallel sections the computation is only executed by the master process.

```
int main (int argc, char **argv ) {
    int P, ...                    // variables declaration and initialization
```
All the variables declared here are duplicated in each process.
All initializations here will apply to all copies of the variables.

```
    SZ_Init(P);                   // initialize message-passing environment
                                  // sets P to number of processes

    ...
```
After call to SZ_Init() only master process executes code, until a parallel section.

```
    SZ_Parallel_begin            // start of parallel section

    …
```
After SZ_Parallel_begin, all processes execute code, until a SZ_Parallel_end

```
    SZ_Parallel_end              // end of  parallel section

    ...
```
Only master process executed code here.

```
    SZ_Finalize();
```
No Suzaku routines here.

```
    return(0);
}
```

Suzaku program structure

Task 1 Point-to-Point Pattern

A sample program called **SZ_pt-to-pt.c** is given below that demonstrates the point-point pattern:

```c
#include <stdio.h>
#include <string.h>
#include "suzaku.h"                              // Suzaku macros

int main(int argc, char *argv[]) {
    int i,j, p, PID;                             //All variables declared here are in every process
    int x = 88,y=99;
    double a[10] = {0,1,2,3,4,5,6,7,8,9};
    double b[10] = {0,0,0,0,0,0,0,0,0,0};
    char a_message[20], b_message[20];
    strcpy(a_message, "Hello world");
    strcpy(b_message, "------------");
    double p=123, q=0;
    double xx[2][3] = {{0,1,2},{3,4,5}},yy[2][3] = {{0,1,2},{3,4,5}};// multidimensional can only be doubles

    SZ_Init(p);    // initialize MPI message-passing environment, sets P to number of processes, not used


    SZ_Parallel_begin                            // parallel section, all processes do this
        PID = SZ_Get_process_num();              // get process ID

        SZ_Point_to_point(0, 1, a_message, b_message);  // send a message from one process to another
        if (PID == 1) printf("Received by process %d = %s\n",PID,b_message); // print it out at destination

        SZ_Point_to_point(0, 1, &x, &y);         // send an int from one process to another
        if (PID == 1) printf("Received by process %d = %d\n",PID,y); // print it out at destination

        SZ_Point_to_point(0, 1, a, b);           // send an array of doubles from one process to another
        if (PID == 1) {                          // print it out at destination
            printf("Received by process %d = ",PID);
            for (i = 0; i < 10; i++)
                printf("%2.2f ",b[i]);
            printf("\n");
        }

        SZ_Point_to_point(0, 1, &p, &q);         // send a double from one process to another
        if (PID == 1)  printf("Received by process %d = %f\n",PID,q);  // print it out at destination

        SZ_Point_to_point(0, 1, xx, yy);         // send an 2-D array of doubles from one process to another
        if (PID == 1) {                          // print it out at destination
            printf("Received by process %d\n",PID);
            for (i = 0; i < 2; i++) {
                for (j = 0; j < 3; j++)
                  printf("%2.2f ",yy[i][j]);
                printf("\n");
            }
        }
    SZ_Parallel_end;                             // end of parallel

    SZ_Finalize();

    return 0;
}
```

Note: As in MPI, all the variables declared here are duplicated in each process. All initializations here will apply to all copies of the variables.

After call to SZ_Init() only master process executes code, until a parallel section.

Only master process executed code here if any.

Do not place any code here (executed by all processes).

Suzaku **SZ_pt-to-pt.c** *program*

SZ_Init(p) is required and sets **p** to be the number of processes, which is determined when you execute the program as in MPI. In this program, **p** is not used. After **SZ_Init(p)**, all code is executed just by the master process, just as in OpenMP a single thread executes the code by default.

SZ_Parallel_begin corresponds to the parallel directive in OpenMP and after it all is code is executed by all the processes. **SZ_Parallel_end** is required to mark the end of the parallel, and includes a global barrier too. After that, the code is again just executed by the master process.

SZ_Get_process_num() returns the process ID and mirrors the **omp_get_thread_num()** routine in OpenMP that gives the thread ID.

SZ_Point_to_point() implements the point-to-point message passing pattern. It has four parameters, the source process ID, the destination process ID, the source array, and the destination array. The source and destination can be individual character variables, integer variables, double variables, or 1-dimensional arrays of characters, integers, or doubles, or multi-dimensional arrays of doubles. The type does not have to be specified. *Multi-dimensional arrays of other types are not currently supported.* The address of an individual variable specified by prefixing the argument the & address operator. This is similar to how one specifies variables in MPI message passing routines. Sending a single number would be inefficient. Sometimes though, it cannot be avoided (for example terminating a loop). Note missing the & is a common programming error.

SZ_Finalize() is required at the end of the program.

You will need the **suzaku. h** file to compile all the Suzaku programs. Make sure **suzahu.h** is placed in same directory as **SZ_pt-to-pt.c**. Compile the **SZ_pt-to-pt.c** program as an MPI program, i.e.:

mpicc -o SZ_pt-to-pt SZ_pt-to-pt.c

(There is a make file you can use also, i.e. **make SZ_pt-to-pt**.)

Execute as an MPI program with two processes:

mpiexec -n 2 ./SZ_pt-to-pt

Save a screenshot of the output. Make sure you understand the program and its output.

What to submit for Task 1

Your submission document should include the following:

1. Screenshot of the **SZ_pt-to-pt.c** executing with two processes.

Task 2 – Matrix Multiplication using Master-Slave Pattern

Previous we did matrix multiplication with MPI using the master-slave pattern. Blocks of rows of matrix **A** is scattered across slaves and matrix **B** is broadcast to all slaves**.** Slaves return a block of rows of C.

A similar program but using Suzaku routines is given as **SZ_matrixmult.c** below. This program demonstrates many of the Suzaku macros (in red).

Slaves (Workers)

Return *blksz rows of* C

C

Broadcast

B

A

Send *blksz* rows of A

Master

Matrix multiplication - master-slave approach

```c
#define N 256
#include <stdio.h>
#include <time.h>
#include <string.h>

#include "suzaku.h"     // Suzaku routines

void print_results(char *prompt, double a[N][N]) {  // available to print out arrays for checking
        int i, j;
        printf ("\n\n%s\n", prompt);
        for (i = 0; i< N; i++) {
                for (j = 0; j < N; j++) {
                        printf(" %.2lf", a[i][j]);
                }
                printf ("\n");
        }
        printf ("\n\n");
}

int main(int argc, char *argv[]) {

        int i, j, k, error = 0;      // All variables declared here are in every process
        double A[N][N], B[N][N], C[N][N], D[N][N], sum;

        double time1, time2;             // for timing
        int P;                           // P, number of processes
        int blksz;                       // used to define blocksize in matrix multiplication

        SZ_Init(P);                  // this initializes MPI environment
                                     // just master process after this
        if (N % P != 0) {
                printf("Error -- N/P must be an integer\n");
        }

        for (i = 0; i < N; i++) {   // set some initial values for A and B
                for (j = 0; j < N; j++) {
                        A[i][j] = j*1;
                        B[i][j] = i*j+2;
                }
        }

        for (i = 0; i < N; i++) {   // sequential matrix multiplication
                for (j = 0; j < N; j++)  {
                        sum = 0;
```

All the variables declared here are duplicated in each process. All initializations here will apply to all copies of the

After call to SZ_Init() only master process executed code, until a parallel

6

```
                for (k=0; k < N; k++) {
                 sum += A[i][k]*B[k][j];
                }
                D[i][j] = sum;
            }
    }

    time1 = SZ_Wtime();    // record  time stamp

    SZ_Parallel_begin

    blksz = N/P;

    double A1[blksz][N];          // used in slaves to hold scattered a
    double C1[blksz][N];          // used in slaves to hold their result

    SZ_Scatter(A,A1);      // Scatter A array into A1

    SZ_Broadcast(B);       // broadcast B array

    for(i = 0 ; i < blksz; i++) {
            for(j = 0 ; j < N ; j++) {
                    sum = 0;
                    for(k = 0 ; k < N ; k++) {
                     sum += A1[i][k] * B[k][j];
                    }
                    C1[i][j] = sum;
            }
    }

    SZ_Gather(C1,C);       // gather results

    SZ_Parallel_end;       // end of parallel, note a barrier here

    time2 = SZ_Wtime();    // record  time stamp

    int error = 0;    // check sequential and parallel versions same answers, within rounding
    for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++)  {
                    if ( (C[i][j] - D[i][j] > 0.001) || (D[i][j] - C[i][j] > 0.001)) error = -1;
            }
    }
    if (error == -1) printf("ERROR, sequential and parallel code give different answers.\n");
    else printf("Sequential and parallel code give same answers.\n");

    printf("elapsed_time = %f (seconds)\n", time2 - time1);  // print out execution time

    SZ_Finalize();
    return 0;
}
```

All the variables declared here are duplicated in each process but scope will be only parallel section. Done here because need to know P first to get size.

Parallel section. All processes executing

After SZ_parallel_end, only master process executed code.

Suzaku matrixmult.c program (master-slave pattern)

The matrices are initialized with values within the program. The sequential and parallel results are checked against each other in the code. The matrix multiplication algorithm implemented is the same as in a previous MPI assignment. Matrix **A** is scattered across processes and matrix **B** is broadcast to all processes**. SZ_Broadcast()**, **SZ_Scatter()**,and **SZ_Gather()** must only be called within a parallel region and correspond to the MPI routines for broadcast, scatter and gather:

SZ_Broadcast(a) broadcasts an array from the master to all processes. **a** is the pointer to the source array in the master and the destination array in all processes (source and destination). Only double arrays ordoubles are allowed. A double variable can also be specified by prefixing the argument with the & address operator.

SZ_Scatter(a,b) scatters an array from the master to all processes. **a** is the source pointer to an array to scatter in the master and **b** is the destination pointer to where data is placed in each process. The size of the block sent to each process is determined by the size of the destination array, **b**. In this case the size of the block sent is blksz rows of N elements where blksz = N/P.

SZ_Gather(a,b) gathers an array from all processes to the master process. **a** is the source pointer to an array being gathered from all processes to the master and **b** is the destination pointer in master where elements are gathered. The size of the block sent from each process is determined by the size of the source array, **a**. In this case the size of the block sent is blksz rows of N elements where blksz = N/P.

SZ_Wtime() simply inserts MPI_Wtime(). Normally this routine would be called only by the master outside a parallel section.

In the current Suzaku implementation, the source and destination arrays must be declared statically or as variable length arrays (i.e. not dynamically). All message passing routines are synchronous for ease of usage (i.e. none the sources or the destinations return until all have completed). There is a barrier at the end of the parallel section so that an explicit barrier is not necessary before the time stamp. (**SZ_Barrier()** is available if one wants a barrier within a parallel section.)

Compile **SZ_matrixmult.c** and execute with four processes. Save a screenshot of the output.

What to submit for Task 2

Your submission document should include the following:

1) Screenshot of the **SZ_matrixmult.c** executing with four processes.

Task 3 Suzaku Monte Carlo π Workpool Program

The workpool pattern is like a master-slave pattern but has a task queue that provides load balancing. Individual tasks are given to the slaves. When a slave finishes a task and returns the result, it is given another task from the task queue, until the task queue is empty. At that point, the master waits until all outstanding results are returned. The termination condition is the task queue empty and all result collected. In Assignment 4, you had to implement the workpool in MPI for the Monte Carlo π calculation. Here the workpool pattern pre-implemented in a very general form.

Workpool Algorithm. In the implementation of the workpool described here (version 1), the data items being sent between the master process and slave processes are limited to 1-D arrays of doubles.[1] The programmer deposes the problem into **T** tasks. Each task consists of a 1-D array of **D** doubles with an associated task ID. Each slave result for a task consists of a 1-D array of **R** doubles with the associated task ID. The master sends out tasks to slaves. Slaves return results and are given new tasks, or a terminator message if there are no more tasks, i.e. if the number of tasks sent reaches **T**. The number of

---

[1] There is an implementation of the workpool called version 2 that closely matches the Seeds interface, using get and put routines, which enables individual data items and arrays to be packed into one task and parts retrieved in any order.

tasks can be less than number of slaves, equal to the number of slaves, or greater than the number of slaves. If the number of tasks is the same as the number of slaves, the workpool becomes essentially a master-slave pattern.

Programmer-written routines. The Suzaku workpool interface is modeled on the Java-based Seeds framework, which we will use in Assignment 6. The programmer must implement four routines:

- **init()** Sets values for the number of tasks (**T**), the number of data items in each task (**D**), and the number of data items in each result (**R**). Called once by all processes at the beginning of the computation.

- **diffuse()** Generates the next task when called by the master.

- **compute()** Executed by a slave, takes a task generated by diffuse and generates the corresponding result.

Slaves   compute()

Master   diffuse()   gather()

Message passing done by framework

- **gather()** Accepts a slave result and develops the final answer. Called by the master.

The workpool is implemented by the routine **SZ_Workpool()**. **init()**, **diffuse()**, **compute()**, and **gather()** are called by **SZ_Workpool()** and given as input function parameters. The application program structure is shown below and consists of the four programmer-written routines and the Suzaku routines.

```
#include <stdio.h>
#include <string.h>
#include "suzaku.h"

void init(int *T, int *D, int *R) {

        ...

        return;
}
void diffuse(int *taskID,double output[D]) {

        ...

        return;
}

void compute(int taskID, double input[D], double output[R]) {

        ...

        return;
}

void gather(int taskID, double input[R]) {
```

```c
        ...

        return;
}

int main(int argc, char *argv[]) {

        int P;                                  // number of processes
        SZ_Init(P);                             // initialize MPI message-passing environment

        SZ_Parallel_begin

                SZ_Workpool(init, diffuse, compute, gather);

        SZ_Parallel_end;

        ...
        printf("Workpool results\n ... ", ....);     // print out workpool results

        SZ_Finalize();

        return 0;
}
```

Suzaku workpool program structure

Suzaku Monte Carlo π workpool program, **MontePi_workpool.c**, is given below.

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h"
                                // required Suzaku constants
#define T 100                   // number of tasks, max = INT_MAX - 1
#define D 1                     // number of data items in each task, doubles only
#define R 1                     // number of data items in result of each task, doubles only

#define S 1000000               // constant used in computation, sample pts done in a slave
double total = 0;               // global variable, final result

void init(int *tasks, int *data_items, int *result_items) {
        *tasks = T;
        *data_items = D;
        *result_items = R;
}
void diffuse(int taskID,double output[D]) {                     // taskID not used in computation
        static int temp = 0;                                   // only initialized first time function called
        output[0] = ++temp;                                    // set seed to consecutive data value
}
void compute(int taskID, double input[D], double output[R]) {
        int i;
        double x, y;
        double inside = 0;

        srand(input[0]);                                       // initialize random number generator
        for (i = 0; i < S; i++) {
                x = rand() / (double) RAND_MAX;
                y = rand() / (double) RAND_MAX;
                if ( ( (x * x + y * y) <= 1.0 ) inside++;
        }
        output[0] = inside;
}
void gather(int taskID, double input[R]) {
        total += input[0];                                     // aggregate answer
```

10

```
}

// additional routines used in this application
double get_pi() {
        double pi;
        pi = 4 * total / (S*T);
        printf("\nWorkpool results, Pi = %f\n",pi);          // print out workpool results
}

int main(int argc, char *argv[]) {
        int i;                          // All variables declared here are in every process
        int P;                          // number of processes, set by SZ_Init(P)
        double time1, time2;        // for timing

        SZ_Init(P);                     // initialize MPI environment, sets P to number of processes

        printf("number of tasks = %d\n",T);
        printf("number of samples done in slave per task = %d\n",S);

        time1 = SZ_Wtime();         // record time stamp
        SZ_Parallel_begin          // start of parallel section
                SZ_Workpool(init,diffuse,compute,gather);
        SZ_Parallel_end;           // end of parallel
        time2 = SZ_Wtime();         // record time stamp

        get_pi();          // calculate final result
        printf("elapsed_time = %f (seconds)\n", time2 - time1);

        SZ_Finalize();
        return 0;
}
```

Suzaku Monte Carlo π workpool program

Workpool code: The workpool routine **SZ_Workpool()** is implemented in **suzaku.c.** It can be compiled with:

mpicc -c -o suzaku.o suzaku.c -lm

to create an object file **suzaku.o** (note the -c option). This avoids having to recompile **suzaku.c** every time you compile application code. **The –lm option is necessary to include the math libraries (also below).**

Application code: **SZ_Workpool**() does not use **suzaku.h** itself but since a workpool needs to be within a parallel section, the application code must include **suzaku.h**. For the commands below, the two files:

suzaku.h
suzaku.o

must be placed in the same directory as the source file. To compile **MontePi_workpool.c**, issue the command:

mpicc -o MontePi_workpool MontePi_workpool.c suzaku.o -lm

or use the provided make file:

make MontePi_workpool

which will compile **suzaku.c** if necessary. **(Add -lm in the make file if missing.)**

Instead of pre-compiling **suzaku.c** into **suzaku.o**, one could also compile both **suzaku.c** and **MontePi_workpool.c** together with:

> **mpicc -o MontePi_workpool MontePi_workpool.c suzaku.c -lm**
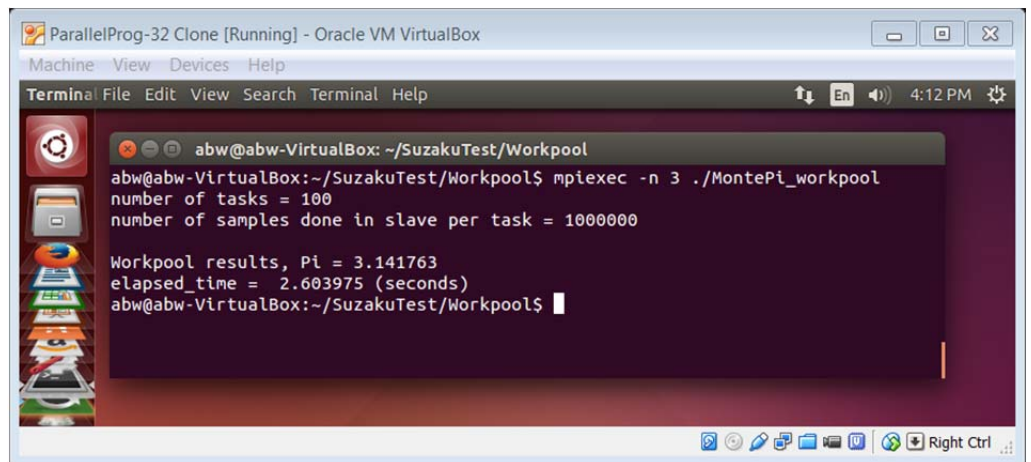
To execute **MontePi_workpool**, issue the command:

> **mpiexec –n <no_of processes> MontePi_workpool**

where **<no_of processes>** is the number of processes you wish to use. The workpool needs at least two processes, master and one slave. Note the master does not act as one slave as in the master-slave pattern because collective routines are not used.

Compile and execute **MontePi_workpool** with *four* processes.

Sample output with three processes

What to submit for Task 3

Your submission document should include the following:
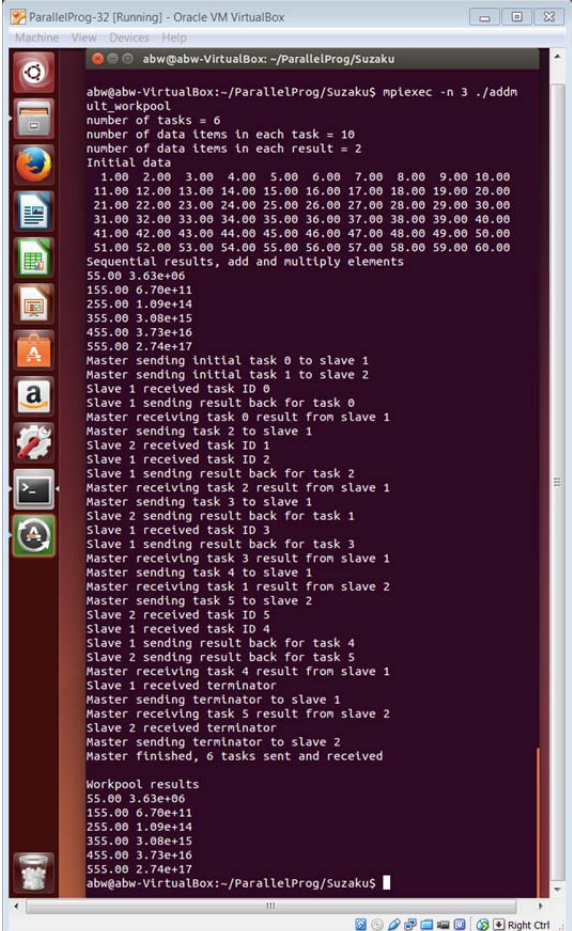
1) Screenshot of **MontePi_workpool** executing on your computer

Task 4 Execute with Debugging Messages

A version of the **SZ_Workpool()** routine is provided that includes print statements to see how the tasks are allocated to slaves and results returned. This version is called **SZ_Workpool_debug()** and can be found in **suzaku.c**.

Rename **SZ_Workpool()** in **MontePi_workpool.c** to **SZ_Workpool_debug()**.

Re-compile and execute with *four* processes.

Sample output with three processes

What to submit for Task 4

Your submission document should include the following:

1) Screenshot of **MontePi_workpool** executing on your computer using the debug version of the workpool, **SZ_Workpool_debug()**.

Task 5 Evaluation of Suzaku (Important, worth 5 points)

Write 1-2 paragraphs on using Suzaku instead on MPI. Describe advantages and disadvantages.

Part 2 Astronomical *N*-Body Problem (35%)

The Problem

Now we will turn to writing our own program. The objective is to find the positions and movements of bodies in space that are subject to gravitational forces from other bodies (e.g., planets) using Newtonian laws of physics. The gravitational force between two bodies of masses m_a and m_b is given by:

$$F = \frac{Gm_a m_b}{r^2}$$

where G is the gravitational constant and r is the distance between the bodies. When there are multiple bodies, each body will feel the influence of each of the other bodies and the forces will sum together (taking into account the direction of each force). Subject to a force, a body will accelerate according to Newton's second law:

$$F = ma$$

where m is the mass of the body, F is the force it experiences, and a is the resultant acceleration. All the bodies will move to new positions due to these forces and have new velocities. Written as differential equations, we have:

$$F = m\frac{dv}{dt}$$

and

$$v = \frac{dx}{dt}$$

where v is the velocity. For a computer simulation, we use values at particular times, t_0, t_1, t_2, and so on, the time intervals being as short as possible to achieve the most accurate solution. Let the time interval be Δt. Then, for a particular body of mass m, the force is given by:

$$F = \frac{m(v^{t+1} - v^t)}{\Delta t}$$

and a new velocity

$$v^{t+1} = v^t + \frac{F\Delta t}{m}$$

where $\boldsymbol{v^{t+1}}$ is the velocity of the body at time $t + 1$, and $\boldsymbol{v^t}$ is the velocity of the body at time t. If a body is moving at a velocity v over the time interval $\Delta \boldsymbol{t}$, its position changes by

$$x^{t+1} - x^t = v\Delta t$$

where $\boldsymbol{x^{t+1}}$ is its position at time $t+1$ and $\boldsymbol{x^t}$ is its position at time t. Once bodies move to new positions, the forces change and the computation has to be repeated. The velocity is not actually constant over the time interval, $\Delta \boldsymbol{t}$, so only an approximate answer is obtained.

If the bodies are in a three-dimensional space, all values (forces, velocities and distances) are vectors and have to be resolved into three directions, x, y, and z. The forces due to all the bodies on each body are added together in each dimension to obtain the final force on each body. Finally, the new position

and velocity of each body are computed due to the forces. This then gives the velocity and positions in three directions. For a simple computer solution, we usually assume a three-dimensional space with fixed boundaries. Actually, the universe is continually expanding and does not have fixed boundaries!

For this assignment, you will use two-dimensional space so the forces, velocities and distances need only be resolved into two directions, x and y.

Two-Dimensional Space. In a two-dimensional space having a coordinate system (x, y), the distance between two bodies at (x_a, y_a) and (x_b, y_b) is given by:

$$r = \sqrt{(x_b - x_a)^2 + (y_b - y_a)^2}$$

The forces are resolved in the two directions, using:

$$F_x = \frac{Gm_a m_b}{r^2}\left(\frac{x_b - x_a}{r}\right)$$

$$F_y = \frac{Gm_a m_b}{r^2}\left(\frac{y_b - y_a}{r}\right)$$

It is convenient to store the position and velocities of the bodies in an array as shown in Table 1. In our case we will only have siz bodies.

Table 1: Input data

Body (Array index)	Mass	Position in x direction	Position in y direction	Velocity in x direction	Velocity in y direction
0					
1					
2					
3					
4					
5					

Sequential Code. The overall gravitational *N*-body computation can be described by the following steps:

```
for (t = 0; t < T; t++) {          // for each time interval

    for (a = 0; a < N; a++) {             // for body a, calculate force on body due to other bodies
        for (i = 0; i < N; i++) {
            if (a != i)   {              // for different bodies
                x_diff = ... ;           // compute distance between body a and body i  in x direction
                y_diff = ... ;           // compute distance between body a and body i in y direction
                r = ... ;                //compute  distance r
```

```
                F = ... ;                    // compute force on bodies
                Fx[a] += ... ;               // resolve and accumulate force in x direction
                Fy[a] += ... ;               // resolve and accumulate force in y direction
            }
        }
    }

    for (a = 0; a < N; a++)  {               // for each body, compute and update positions and velocity
        A[a][x_velocity]= ...  ;             // new velocity in x direction, column 4 in Table 1
        A[a][y_velocity]= ...  ;             // new velocity in y direction, column 5 in Table 1
        A[a][x_position] = ...  ;            // new position in x direction, column 2 in Table 1
        A[a][y_position] = ...  ;            // new position in y direction, column 3 in Table 1
    }

} // end of simulation
```

Task 1

Write a sequential C program that computes the movement on N bodies in two dimensions, where N is a constant. Set $N = 6$. The initial data is to be hardcoded into the program for simplicity and consists of an array of doubles, **A[6][5]** holding the mass, initial positions and velocities of six bodies as follows:

Mass	x pos	y pos	x vel	y vel
25.0	400.0	400.0	0.0	0.0
20.0	200.0	400.0	3.0	4.0
30.0	50.0	600.0	1.0	0.0
50.0	400.0	200.0	1.0	0.0
40.0	700.0	700.0	-1.0	0.0
70.0	200.0	100.0	-1.0	0.0

The values are not actual values in a real simulation. Set the gravitational constant to be 100. Set the number of time intervals, T, and the time interval Δt through keyboard input. Use an x-y resolution of 1000 x 1000 points. Update the data array after each iteration and display the final values computed at the end of the computation.

If two bodies get very close, the forces will tend to infinity. Hence under that situation, delete both bodes from the table so as to represent the bodies being destroyed. In the program, do this when the distance is less than the sum of the masses divided by 2. (Strictly the diameter of a body is proportional to the cube root of the mass.)

Demonstrate your program with a time interval of 0.05 (Δt) with 1000 iterations (T) and 10000 iterations (T). Also give one sample result with your own values. It is suggested that you use a make file to compile the program. A sample is provided. This becomes more convenient later.

Overleaf is sample output with 0.05 (Δt) and 1000 iterations (T)[2]:

---

[2] This particular program actually read the input data from a file.

16

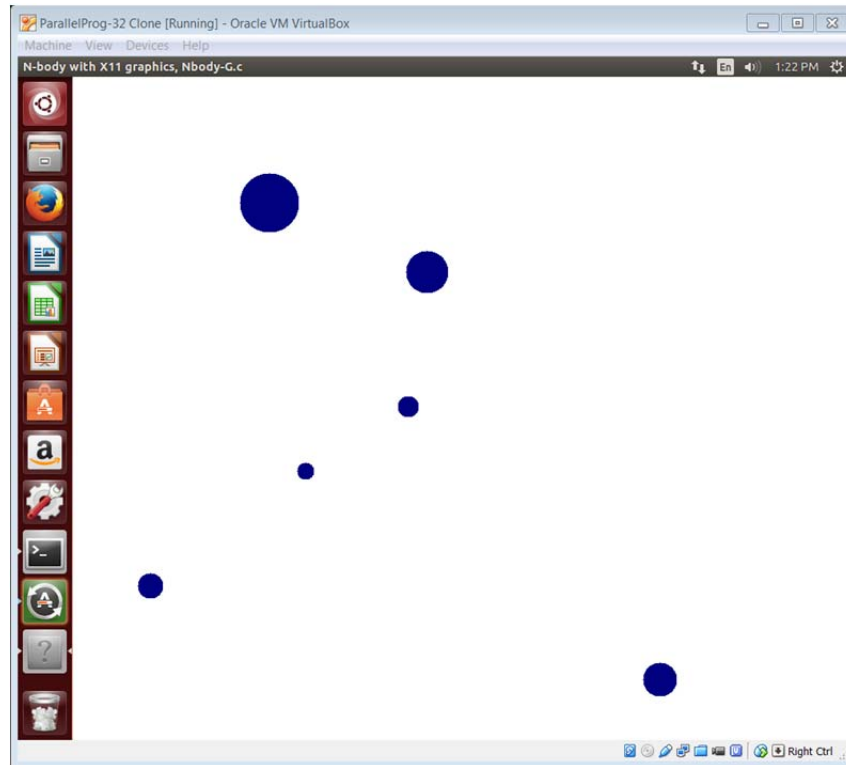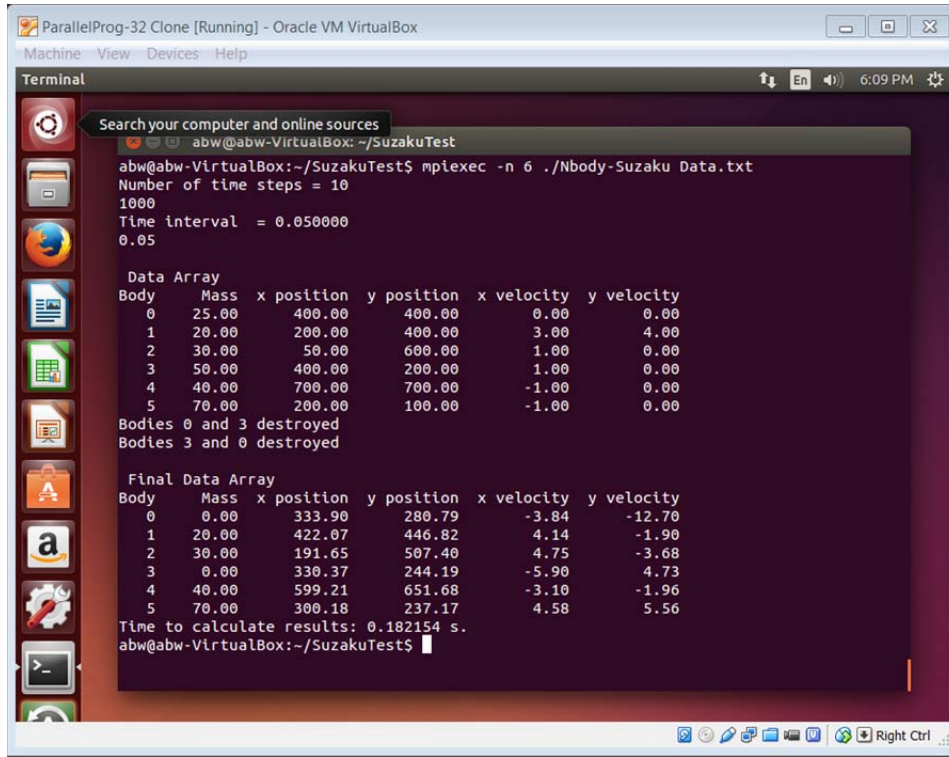Notice in this particular input, bodies 0 and 3 are destroyed.

What to submit for this task

Your submission document should include, *but is not limited to*, the following:
1. Code listing of your sequential *N*-body program
2. Screenshot of the sequential *N*-body program executing with the specified data using a time interval of 0.05 with 1000 and 10000 iterations.
3. Another sample output.

Task 2 Display movement of the bodies over time

Add code to display the movement of the bodies while program is executing using X11 graphics. Refer to the separate notes on creating X11 graphical output. Take advantage of the provided header file **X11Macros.h**. Sample output is given below:

What to submit from this task

Your submission document should include, *but is not limited to*, the following:

1. Code listing of your sequential *N*-body program with the X-11 graphics
2. Three screenshots of the program executing producing graphical output with the specified data using a time interval of 0.05 and 1000 iterations. (Video better)
3. Another sample output.

Part 3 Converting the *N*-Body program into a Master-Slave Suzaku MPI program (30%)

Modify your *N*-body program to be a master-slave Suzaku program that uses six processes, one for each body. Incorporate code to measure the time of execution. *You do not need incorporate the X11 code although you can if you wish (no extra credit).* The numerical output has to correct and match that of the sequential program. . There may be small rounding differences to the sequential version. It highly recommended that you write and test this program in stages, as it easy to make mistakes! Note the code will not function correctly if you use more than 6 processes and with less processes, less bodies are considered. Sample output is shown below:

In the parallel version, we also see two processes each deleting the bodies, one assigned to body 3 and one assigned to body 5, without altering the code from the sequential version. In actuality, a process can only delete its body from the table as it only returns to the master one row of the table, that body associated with its body.

Some notes and suggestions:

1. Only the master process should read the keyboard. All processes will need this data. Hence you will need to broadcast the keyboard input data to all processes. Note the Suzaku broadcast now can handle multiple data types without casting. You will also need to broadcast the initial data array.

2. Suzaku does not allow a master process to nest a parallel section. When the parallel section begins, the master section automatically ends. Hence one cannot have loop in the master process which includes a parallel section.

3. Each process in the parallel region will be responsible for one body. You will need **SZ_Get_process_num()** to get the process ID. Hence the loop with the *a* variable in the sequential code template given earlier will not exist.

4. A **SZ_AllBroadcast()** will be needed to broadcast the new data values at the end of each iteration. This routine broadcasts the *i*th row of an array of doubles from the *i*th process to every other process, for all *i*. (This is not the same as an MPI_Allgather().) Assumes there are *P* rows in the array.

5. A **SZ_Barrier()** should not be needed as all the low-level Suzaku macros are now synchronous.

You must not use any explicit MPI routines, only Suzaku routines to create MPI code.

What to submit for Part 3

Your submission document should include the following:

1) A listing of your Suzaku program for the *N*-body problem
2) Screenshot of the program executing with the specified data using a time interval of 0.05 with 1000 iterations and 10000 iterations.
3) Screenshot of the output of sequential version with same inputs to prove that the parallel version does in fact create the same results.
4) A brief evaluation of using the Suzaku macros comparing and contrasting using Suzaku with using MPI. Describe your experience and opinions, and any suggestions for improvement. (5 points)

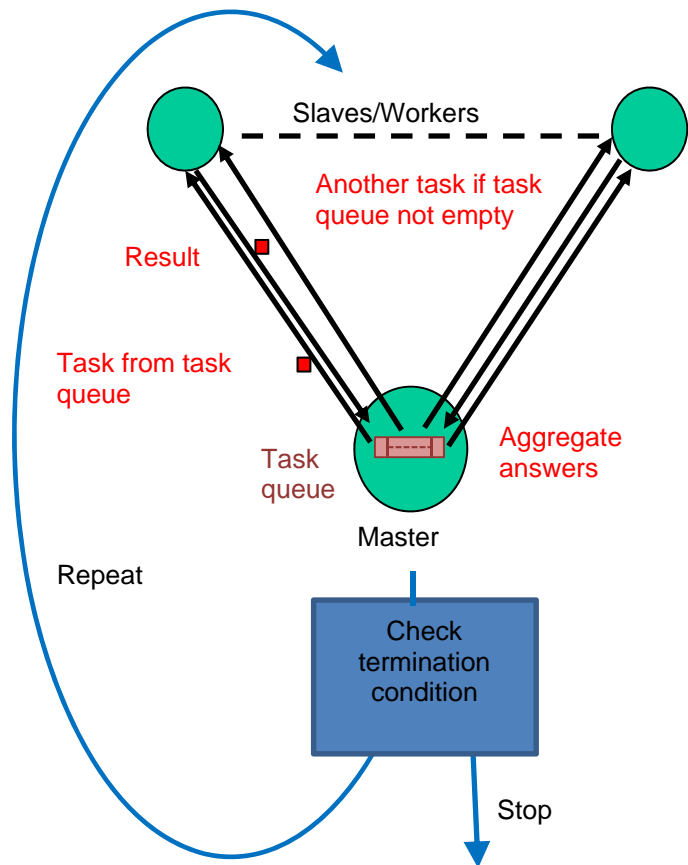# Part 4 *N*-Body Problem Using a Workpool (15% extra credit)

In this part you are asked to reformulate the astronomical *N*-Body program of Part 2 as a Suzaku workpool. Each task consists of the information of one body (mass/position/velocity). A slave receives one body to work on and returns updated information for that body. The workpool itself is inside a for loop and repeated for each time step. Hence we now have a "synchronous iterative pattern." Because synchronous iterative patterns are common, such combined patterns can be provided pre-implemented more efficiently than using a for loop inserted by the programmer. The pre-implemented synchronous iterative Stencil pattern appears in Seeds and in Paraguin.

Synchronous iterative workpool

Re-write your the astronomical *N*-Body program of Part 2 to be a Suzaku workpool.

The *N*-Body program has keyboard input for number of time steps and time interval as in the original program. These values have to be broadcast to all processes before the workpool. The end of the workpool is a synchronization point with an implicit barrier because of the way the workpool works. The master does not stop until all the slave processes finish first. It is not necessary to have the same number of slaves as bodies as in the programs in master slave version. Slaves are not assigned to particular bodies and it is possible for one slave to process all the bodies. As the order that the positions and velocities of bodies are updated is indeterminate, one cannot update the data array with new positions and velocities as they are calculated as that information could be used with other bodies in the same iteration. One could use two arrays but to avoid copying from one array to the other array at the end of each time interval, we use a technique you saw when solving the Heat equation - merging the two arrays into a three dimensional array **A[N]N][2]** and alternating the third index between 0 to 1 at each iteration. Here this is done in the **init**() routine, called at the beginning of the workpool.
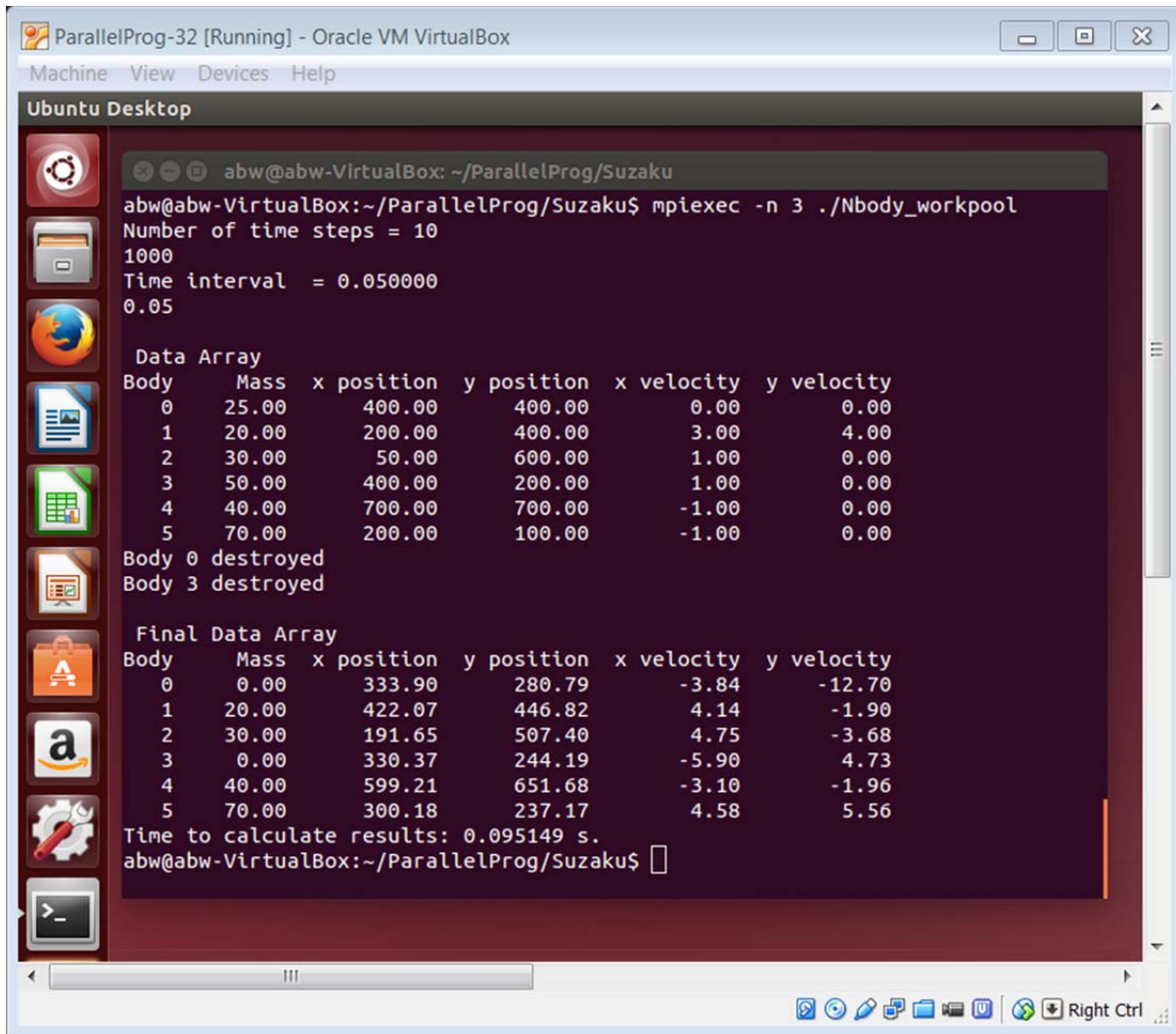
Compile and execute your program with different numbers of three processes, including three processes

Try executing with debug messages and multiple times to see how slaves are allocated. Try different numbers of processes.

Sample output (Note number of processes does not have to be the same as number of bodies.)

What to submit for Part 4

Your submission document should include the following:

1) Program listing of your workpool *N*-body program.

2) Screenshot of your workpool *N*-body program executing on your computer using three processes and the regular version of **SZ_Workpool()**.

3) Screenshot of your workpool *N*-body program executing on your computer using three processes and the debug version of **SZ_Workpool()**.

4) Discussion of the results.

Assignment Report Preparation and Grading

Produce a report that shows that you successfully followed the instructions and performs all tasks by taking screenshots and include these screenshots in the document.

Every task and subtask specified will be allocated a score so make sure you clearly identify each part/task you did. Make sure you include everything that is specified in the "Include ..." section at the end of each task/part as a minimum. **Include all code, not as screen shots of the listings but complete properly documented code listing.** The programs are often too long to see in a single screenshot and also not easy to read if small. Using multiple screenshots is not option for code. Copy/paste the code into the Word document or whatever you are using to create the report.

You will lose 10 points if you submit code as screenshots.

Assignment Submission

Convert your report into a *single pdf* document and submit the pdf file on Moodle by the due date as described on the course home page. It is extremely important you submit only a pdf file.

You will lose 10 points if you submit in any other format (e.g. Word, OpenOffice, ...). Submitting a zip with multiple files is especially irritating.