SUZAKU Pattern Programming Framework Specification

1 - Structure and Low-Level Patterns

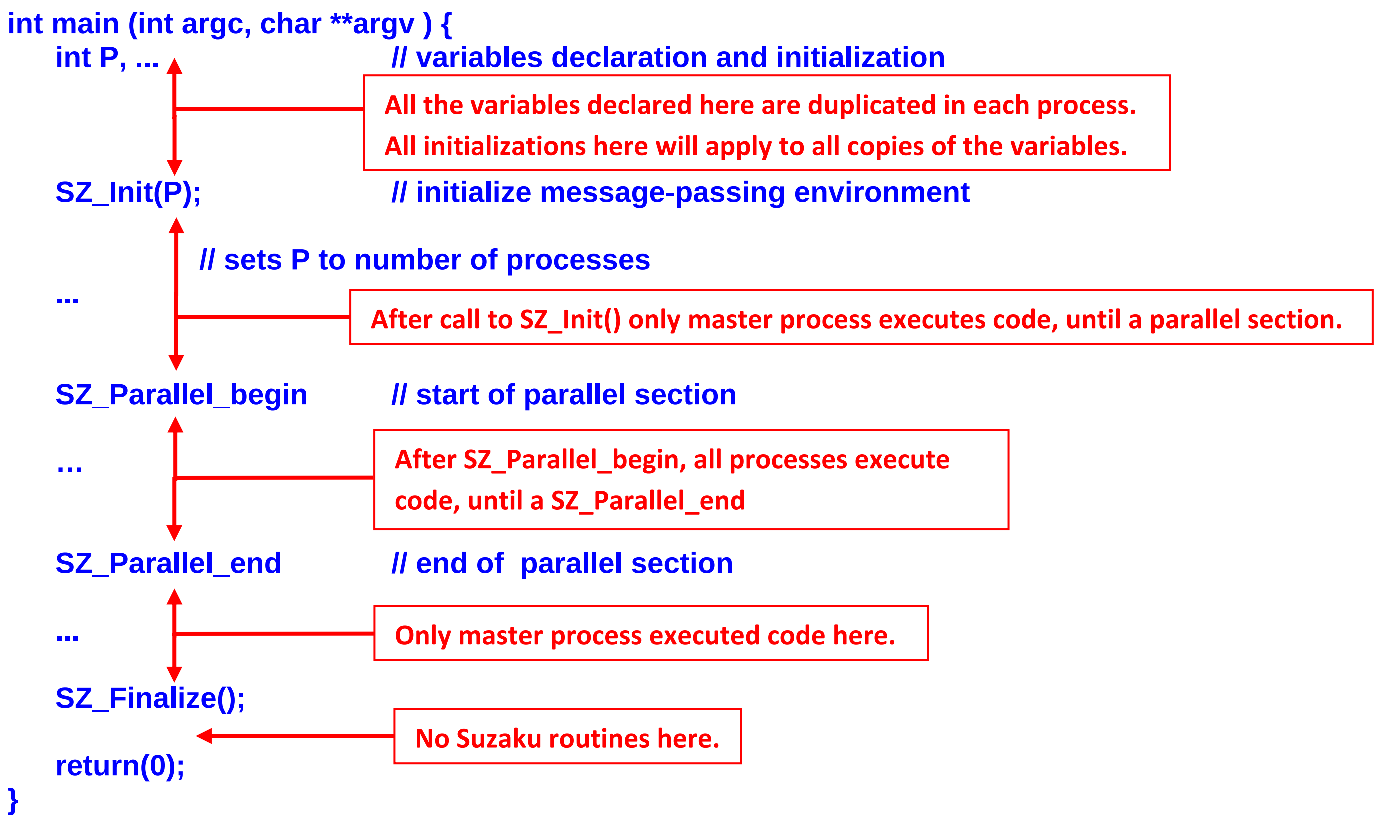
B. Wilkinson, March 17, 2016.

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 workpool. Suzaku is still under development. Several unique patterns and features are being provided 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. To use Suzaku, you must have an implementation of MPI installed. OpenMPI is recommended. This document describes the application program structure, the low level message passing routines, and the workpool pattern.

Basic Structure and Low Level Routines

1.1 Program structure

The computational model is similar to OpenMP but using processes instead of threads. With the process-based model, there is no 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
    SZ_Init(P);                 // initialize message-passing environment
                      // sets P to number of processes
    ...
    SZ_Parallel_begin           // start of parallel section
    ...
    SZ_Parallel_end             // end of  parallel section
    ...
    SZ_Finalize();

    return(0);
}
```

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.

After SZ_Parallel_begin, all processes execute code, until a SZ_Parallel_end

Only master process executed code here.

No Suzaku routines here.

1.2 Program Structure Routines

Initialization

SZ_Init(int P);

Purpose: To be used to initialize the message passing environment and declare variables used by Suzaku internally. No Suzaku routines must be placed before **SZ_Init(P)**. **SZ_Init(P)** is required and sets **P** to be the number of processes in each process. 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. All processes have a process ID, an integer from 0 to P - 1. The master ID is 0.

Parameter:

P Name of integer variable used to store number of processes. Must be declared by the programmer for each process before SZ_Init().

Assumptions and limitations: **argc** and **argv** must be declared in **main()**. The number of processes is set on the command line when executing the program using the MPI command **mpiexec** and read from the command line. As a message-passing model, there are no shared variables. All variables are local to a process, and generally should be declared before **SZ_Init(P)**. p is an output parameter but does not need the & address operator because implementation as a macro (inline substitution).

Finalize

SZ_Finalize();

Purpose: To be used at the end of the program to close MPI. **SZ_Finalize()** is required. No Suzaku routines must be placed after **SZ_Finalize()**.

Assumptions and Limitations: All processes still exist after **SZ_Finalize()** and any code placed after SZ_Finalize() will be executed by all processes. Typically one does not want to do this so do not place any call after **SZ_Finalize()**. Do not call any Suzaku routines after **SZ_Finalize()**. Any MPI-based code such as Suzaku routines will not execute and will cause an error condition. (). This is the same as with **MPI_Finalize()**.

Parallel Section

```
SZ_Parallel_begin
        ...
SZ_Parallel_end;
```

Purpose: Used to indicate code executed by all processes. **SZ_Parallel_begin** corresponds to the parallel directive in OpenMP and after it all code is executed by all the processes.

SZ_Parallel_end is required to mark the end of the parallel, and includes a global barrier to match a parallel section in OpenMP without a no-wait clause. After that, the code is again just executed by the master process.

Limitations: Multiple parallel sections are allowed. However a master process cannot nest a parallel section. When the parallel section begins, the "master section" automatically ends. Hence the scope of any variable declared after **SZ_Init()** and before a parallel section ends at the **SZ_Parallel_begin**. If you want a variable to have the scope of all master sections, declare it before **SZ_Init()**. Similarly one cannot have a loop or structured block in the master section that includes a parallel section.

1.3 Runtime Environment

Process ID

SZ_Get_process_num();

Purpose: Returns the process ID and mirrors the **omp_get_thread_num()** routine in OpenMP, which gives the thread ID. Processes are numbered from 0 to P - 1 where there are P processes, with the master process having number zero.

1.4 Low-Level Patterns

Patterns are created within a parallel section. The following low-level patterns implemented so far:

- Point-to-point pattern
- Broadcast (master to all slaves)
- All-to-All Broadcast (all slaves to all slaves)
- Scatter (from master to slaves)
- Gather (from slaves to master)
- Master-slave pattern

Point-To-Point Pattern

SZ_Point_to_point(p1, p2, a, b);

Purpose: Sends data from one process to another process.

Parameters:

p1	Source process ID
p2	Destination process ID
a	Pointer to the source array
b	Pointer to destination array

Limitation: 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.

Broadcast Pattern

SZ_Broadcast(double a);

Purpose: To broadcast an array from the master to all processes.

Parameter:

a Pointer to the source array in the master and the destination array in all processes (source and destination)

Limitation: 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 feature has been added as sometimes it is necessary to send a single value, but it is inefficient and should be avoided if possible.

All-to-All Broadcast Pattern

SZ_AllBroadcast(double a)

Purpose: To broadcast the *i*th row of a 2-D array from the *i*th process to every other process, for all *i*.[1]

Parameters:

a Pointer to array (source and destination)

Limitation: The array must be a two-dimensional array of doubles. Assumes there are *P* rows in the array.

Scatter Pattern

SZ_Scatter(double a, double b);

Purpose: To scatter an array from the master to all processes, that is, to send consecutive blocks of data in an array to consecutive destinations. The size of the block sent to each process is determined by the size of the destination array, **b**. Typically used with 2-D arrays sending one or more rows to each process.

Parameters:

a Source pointer to an array to scatter in the master.
b Destination pointer to where data is placed in each process.

[1] This is not the same as an MPI_Allgather(). In MPI_Allgather(), the block of data sent from the *i*th process is received by every process and placed in the *i*th block of the receive buffer.

Limitation: The source and destination arrays must be arrays of doubles in the current implementation. The source and destination can be the same if the underlying MPI implementation allows that (as in OpenMPI but not MPICH).

Gather Pattern

```
SZ_Gather(double a, double b);
```

Purpose: To gather an array from all processes to the master process, that is, to collect a block of data from all processes to the master placing the blocks in the destination in the same order as the source process IDs. This operation is the reverse of scatter. The size of the block sent from each process is determined by the size of the source array, **b**. Typically used with 2-D arrays receiving one or more rows from each process.

Parameters:

a	Source pointer to an array being gathered from all processes to the master.
b	Destination pointer in master where elements are gathered.

Limitation: The source and destination arrays must be arrays of doubles in the current implementation. The source and destination can be the same if the underlying MPI implementation allows that (as in OpenMPI but not MPICH).

Master Process

```
SZ_Master
    <Structured block>
```

Purpose: To be used to indicate code only executed only by the master process (within a parallel section). Must be followed by the code to be executed by master as a single statement or a structured block, e.g.:

```
SZ_Master {

    ...             // code executed by master only

}
```

The opening parenthesis can be on the same line or the next line.[2]

Specific Process

```
SZ_Process(PID)
    <Structured block>
```

[2] OpenMP directives require the opening parenthesis to be on the next line.

Purpose: To be used to indicate code only executed only by a specific process (within a parallel section). Must be followed by the code to be executed by master as a single statement or a structured block, e.g.:

```
SZ_ Process(PID) {

        ...                 // code executed by specific process only

}
```

The opening parenthesis can be on the same line or the next line.

Parameter:

PID ID of process that is to execute structured bock, as obtained from **SZ_Get_process_num();**

Note: **SZ_Process(0)** is the same as **SZ_Master**. **SZ_Process()** might be useful for testing and debugging but in general it is recommended that one should avoid using **SZ_Process()** as it does not conform to the concept of using the pattern approach and leads to unstructured programming.

Master-Slave Pattern

The master-slave pattern can be implemented in Suzaku using broadcast, scatter, and gather patterns. For efficient mapping to collective MPI routines, the master also acts as one of the slaves. The function that the slaves execute is placed after the scatter and broadcast and before the gather. For example, matrix multiplication might look like:

```
SZ_Parallel_begin

    SZ_Scatter(a,c);            // Scatter A array
    SZ_Broadcast(b);            // broadcast B array

    … // compute function, block matrix multiplication. Programmer implements routine

    SZ_Gather(b,a);             // gather results

SZ_Parallel_end;
```

Complete sample programs are given later.

Synchronization and Timing

The following routine can be used within a parallel section:

Barrier

```
    SZ_Barrier();
```

Purpose: Waits until all processes reach this point and then returns. Process synchronization is implicit in message-passing routines, but occasionally one wants to create a synchronization point.

Timing

```
SZ_Wtime();
```

Purpose: To provide time stamp. Returns the number of seconds since some time in the past (a floating-point number representing wallclock). Simply substitutes **MPI_Wtime()**. It is expected that this routine would be called only by the master process outside a parallel section

Sample usage:

```
double start, end;

start = SZ_Wtime();
SZ_Parallel_begin
        ...                         // to be timed
SZ_Parallel_end;
end = SZ_Wtime();

printf("Elapsed time =  %f seconds\n", end - start);
```

1.5 Implementation Limitations of Low-Level Routines

1. *Use of macros.* Macros are currently used to implement the low level routines described so far to avoid needing to specify the data type and size. Macros perform in-line text substitution and substitute the formal parameter with the provided arguments without regard to type or any implied meaning before compilation. Great care is needed with macros as there are situations in which in-line substitution will not work. Most of the message passing macros have been written to allow them to be placed anywhere a single statement could be placed but none of macros must be used in the body of if, if-else or other control constructs if it is possible not all the processes execute the code. In general, it is best to avoid placing any Suzaku macros or routines inside control constructs. Interestingly the MPI standard allows implementers to implement a few specific MPI routines as macros.

2. *Variables names:* Programmer cannot use a variable name starting with **__sz** (two underscores sz) because the macros perform in-line substitution of code and use these variable names. The higher-level compiled routines described later do not have this limitation.

3. *Macro Arguments*

   Mostly arguments are specified as pointers. For an array that would simply be the name of the array. Single variables can be specified by prefixing the variable name with the & address operator, or a one-element array could be used. Sending a single data item would be inefficient but is sometimes necessary, and is allowed with a single variable prefixed with the & address operator or with the use of a one-element array. To send multiple variables, it is recommended to pack individual values into an array for transmission to another process.

4. *Size of Arrays:*

The macros use **sizeof()** to determine the size of array arguments. All arrays being sent between processes must be declared in such a way that the size of the array can be obtained using **sizeof()**. Hence the arrays cannot be created dynamically using **malloc.** Generally declare arrays statically where their size is known at compile time, e.g. **double A[N];** where **N** is a defined constant. C allows "variable length arrays" to be declared where the size is specified as a variable, for example **double A[x];** where **x** is previously declared and assigned values. The size of variable length arrays can be returned with **sizeof()** but variable length arrays have limitations. For example the maximum size is more limited as the arrays are stored on the stack and static storage allocation using the **static** keyword is not allowed and variable length arrays are not allowed at file scope. However sometimes variable length arrays will be necessary. An example using variable length arrays is given the matrix multiplication code given later.

5. *Data Types:*

To make the implementation simple, in many cases the data being sent between processes must be doubles (variables or arrays of any dimension). **SZ_Point_to_point()** and **SZ_Broadcast()**) also allow a wide range of other types for added flexibility and the likelihood that other types may be needed - characters, integers, doubles, 1-dimensional arrays of characters, 1-dimensional arrays of integers, 1-dimensional arrays of doubles, and multi-dimensional arrays of doubles. The type and size does not have to be specified. Multi-dimensional arrays of other types are not currently supported. Floats are not supported at all.

6. *Synchronization:*

The implementation of all Suzaku low-level message passing routines now have been made synchronous for ease of use, that is, all the processes involved do not return until the whole operation has been completed. This is not the same as the MPI. There is some confusion in the literature on this matter as the MPI standard does not define its implementation and it is possible that a particular implementation is more constraining than the standard. The basic MPI point-to-point and collective routines do not necessarily synchronize processes. Each process will return when their local actions have completed ("locally blocking"). This means that the point-to-point routine will return in the source when the message has left the source process but the message may not have reached the destination. It does allow the programmer to alter the values of the variables used as augments in the source process though. The destination process returns when the message has been received and similarly the programmer to alter the values of the variables used as augments in the destination process. MPI does offer synchronous versions of point-to-point message passing that are used here, and in fact even when MPI programmers use the local blocking routines, there must allow for the possibility that they will operate in synchronous fashion. The MPI collective routines also are non-blocking. Each process will return when it has completed its local actions. In Suzaku, a barrier is added to force all the processes to wait to each other as MPI does not offer synchronous collective routines.

7. *Software needed.* To use Suzaku, you must have an MPI environment installed. We use OpenMPI.

8. *Printing*

Printing output generated by different processes can be a challenge. Although standard output is redirected to the master process, when the output would appear is indeterministic generally and the output individual processes might appear if different orders. A single printf output any one process will not be disturbed once it starts, that is the individual characters of the printf buffer will not be interleaved with those of another printf of another process, but the complete lines might be interleaved. One solution to make sure the printout of an array is keep together and ensuring the output is in process order is shown below:

```
PID = SZ_Get_process_num();          // get process ID
for (i = 0; i < P; i++) {
        if (i == PID) {
            printf("Received by process %d \n",PID);
            for (j = 0; j < 10; j++)
                printf("%5.2f ",A[j]);   // print it out at destination
        }
        SZ_Barrier();
}
```

1.6 Compilation and Execution of Low-Level Routines

To use Suzaku, you must have an MPI installed. We use and recommend OpenMPI. Currently the low-level message passing patterns described in this document are implemented with macros placed in **suzaku.h**. The programmer must include the **suzaku.h** file to use the Suzaku macros, i.e.:

```
#include "suzaku.h"                    // Suzaku macros
...
int main (int argc, char **argv ) {
...
return(0);
}
```

Here, the **suzaku.h** file must be in the same directory as the main source program. **argc** and **argv** must be provided as main parameters for MPI. To compile a program **prog1** containing suzaku macros, simply compile as an MPI program, i.e., execute the command:

```
mpicc -o prog1 prog1.c
```

mpicc uses **gcc** to links libraries and create the executable, and all the usual features of **gcc** can be used.

To execute prog1, issue the command:

```
mpiexec -n <no_of processes> ./prog1
```

where <**no_of processes**> is the number of processes you wish to use.

Sample Programs with Suzaku routines

1 Point-to-Point Pattern

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

```
// B. Wilkinson Nov 14, 2015 Testing SZ_Point_to_point with different types
#include <stdio.h>
#include <string.h>
#include "suzaku.h"               // Suzaku macros
int main(int argc, char *argv[]) {
    int i,j, P, PID;
    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,
```

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

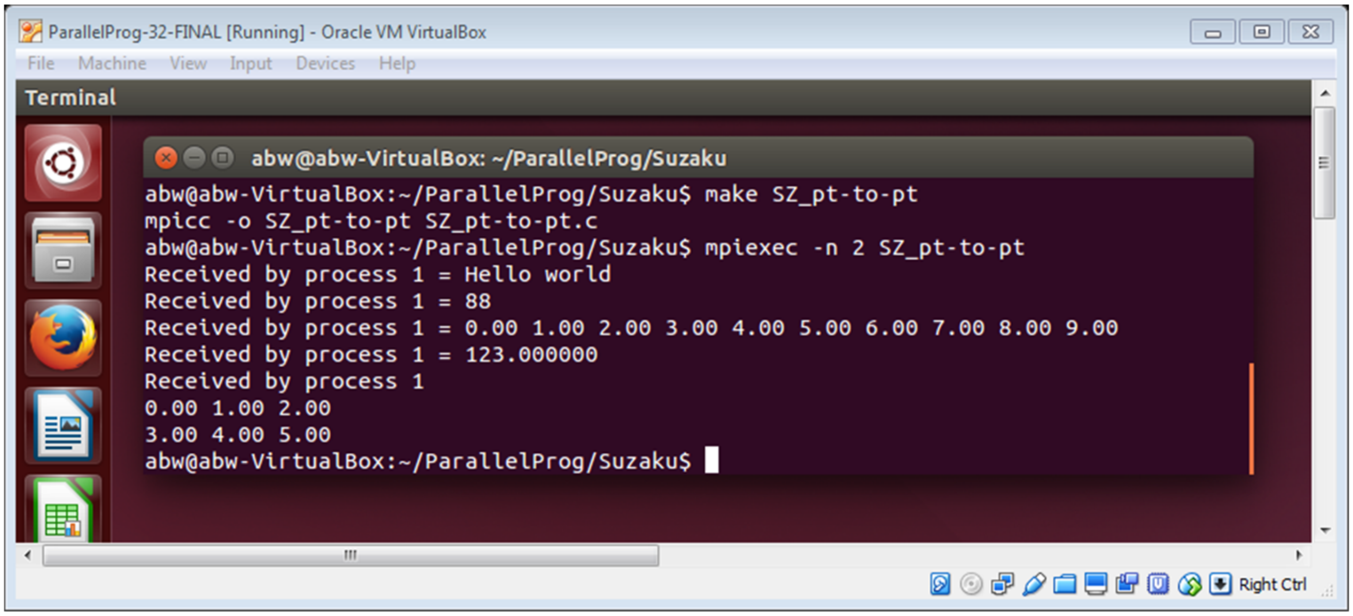
After call to SZ_Init() only master process executes code, until a parallel section.

```
    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;
}
```

Only master process executed code here.

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

Sample output:


ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
abw@abw-VirtualBox: ~/ParallelProg/Suzaku
Right Ctrl

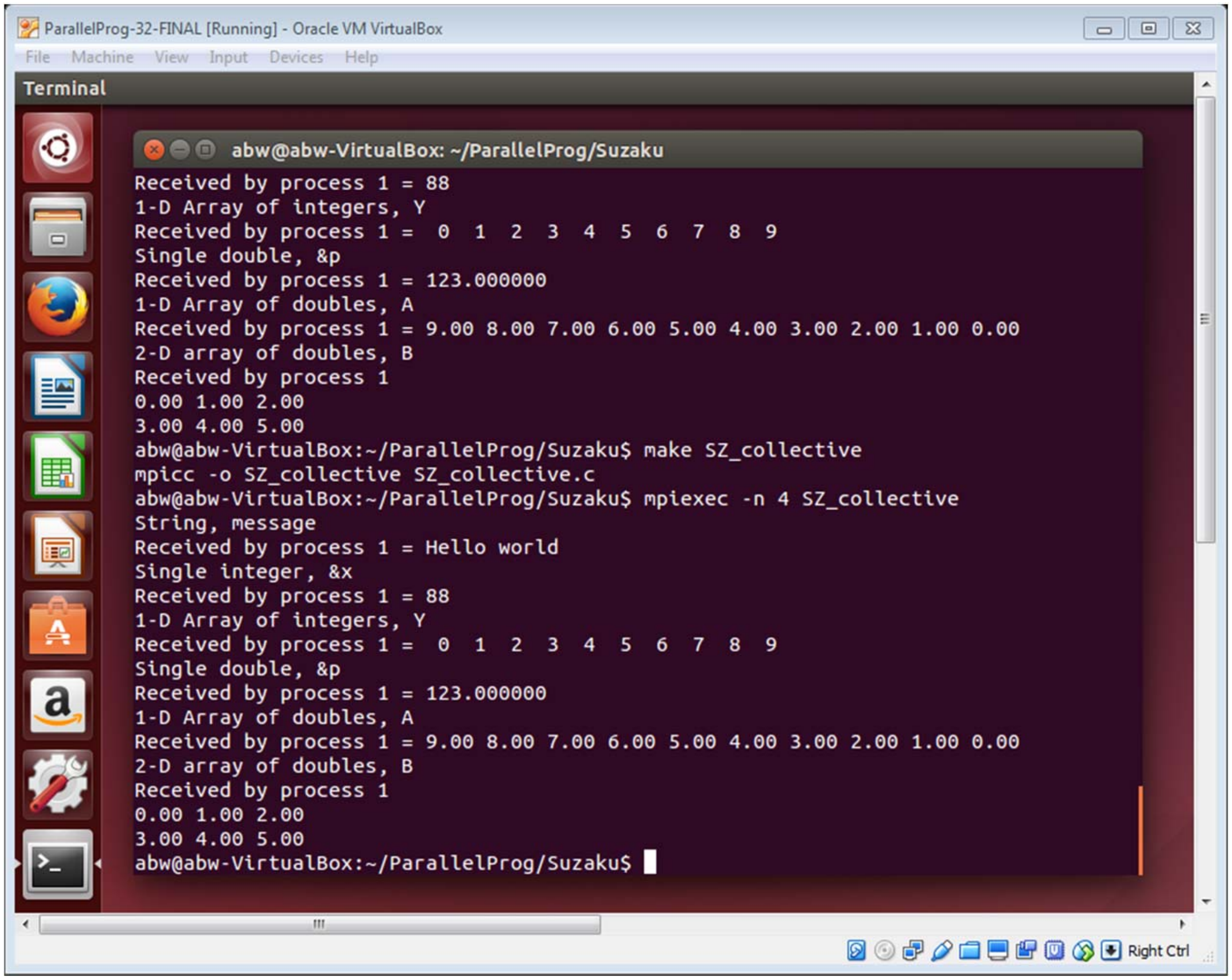
```
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ make SZ_pt-to-pt
mpicc -o SZ_pt-to-pt SZ_pt-to-pt.c
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpiexec -n 2 SZ_pt-to-pt
Received by process 1 = Hello world
Received by process 1 = 88
Received by process 1 = 0.00 1.00 2.00 3.00 4.00 5.00 6.00 7.00 8.00 9.00
Received by process 1 = 123.000000
Received by process 1
0.00 1.00 2.00
3.00 4.00 5.00
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

2. Broadcast

A sample program called **SZ_collective.c** is given below that demonstrates the Suzaku broadcast macro with various data types. Note the data type and the size need not be specified in the code.

```
// B. Wilkinson Dec 28, 2015 Testing SZ_Broadcast with different types
#include <stdio.h>
#include <string.h>
#include "suzaku.h"                           // Suzaku macros

int main(int argc, char *argv[]) {
        char message[20];
        int i,j,k, P, PID;                            //All variables declared here are in every process
        int x = 0;
        int Y[10] = {0,0,0,0,0,0,0,0,0,0};
        double p=123;
        double A[10] = {0,0,0,0,0,0,0,0,0,0};
        double B[2][3] = {{0,0,0},{0,0,0}};           // multidimensional can only be doubles
        strcpy(message, "------------");

        SZ_Init(P);                                           // initialize MPI message-passing environment,
                                                              // Initialize, only master does this until parallel section
        strcpy(message, "Hello world");
        x = 88;
        for (i = 0; i < 10; i++) {
                Y[i] = i;
                A[i] = 9 - i;
        }
        p=123;
        k = 0;
        for (i = 0; i < 2; i++)
          for (j = 0; j < 3; j++)
                B[i][j] = k++;

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

                SZ_Broadcast(message);                        // broadcast a message
                if (PID == 1) printf("String, message\nReceived by process %d = %s\n",PID,message); // print at  dest.

                SZ_Broadcast(&x);                             // broadcast an int
                if (PID == 1) printf("Single integer, &x\nReceived by process %d = %d\n",PID,x); // print at  dest.

                SZ_Broadcast(Y);                              // broadcast an array of doubles
                if (PID == 1) {                               // print at  dest.
                        printf("1-D Array of integers, Y\nReceived by process %d = ",PID);
                        for (i = 0; i < 10; i++)
                                printf("%2d ",Y[i]);
                        printf("\n");
                }

                SZ_Broadcast(&p);                             // broadcast a double
                if (PID == 1)  printf("Single double, &p\nReceived by process %d = %f\n",PID,p);      // print at  dest.

                SZ_Broadcast(A);                              // broadcast an array of doubles
                if (PID == 1) {                               // print at  dest.
                        printf("1-D Array of doubles, A\nReceived by process %d = ",PID);
                        for (i = 0; i < 10; i++)
                                printf("%2.2f ",A[i]);
                        printf("\n");
                }

                SZ_Broadcast(B);                              // broadcast a 2-D array of doubles
                if (PID == 1) {                                // print at  dest.
                        printf("2-D array of doubles, B\nReceived by process %d\n",PID);
                        for (i = 0; i < 2; i++) {
                           for (j = 0; j < 3; j++)
                                printf("%2.2f ",B[i][j]);
                           printf("\n");
                        }
                }
        SZ_Parallel_end;                                        // end of parallel
        SZ_Finalize();
        return 0;
}
```

Sample output


ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal
abw@abw-VirtualBox: ~/ParallelProg/Suzaku
Right Ctrl

```
Received by process 1 = 88
1-D Array of integers, Y
Received by process 1 =  0  1  2  3  4  5  6  7  8  9
Single double, &p
Received by process 1 = 123.000000
1-D Array of doubles, A
Received by process 1 = 9.00 8.00 7.00 6.00 5.00 4.00 3.00 2.00 1.00 0.00
2-D array of doubles, B
Received by process 1
0.00 1.00 2.00
3.00 4.00 5.00
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ make SZ_collective
mpicc -o SZ_collective SZ_collective.c
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpiexec -n 4 SZ_collective
String, message
Received by process 1 = Hello world
Single integer, &x
Received by process 1 = 88
1-D Array of integers, Y
Received by process 1 =  0  1  2  3  4  5  6  7  8  9
Single double, &p
Received by process 1 = 123.000000
1-D Array of doubles, A
Received by process 1 = 9.00 8.00 7.00 6.00 5.00 4.00 3.00 2.00 1.00 0.00
2-D array of doubles, B
Received by process 1
0.00 1.00 2.00
3.00 4.00 5.00
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

3. Matrix Multiplication

A sample program called **SZ_matrixmult.c** is given below that demonstrates many of the Suzaku macros and variable length arrays (A1 and C1).

```
#define N 256
#include <stdio.h>
#include <time.h>
#include "suzaku.h"                              // Suzaku routines

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) {
                error = -1;
                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;
                        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;
                  }
```

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 executed code, until a parallel section.

Parallel section

All processes executing

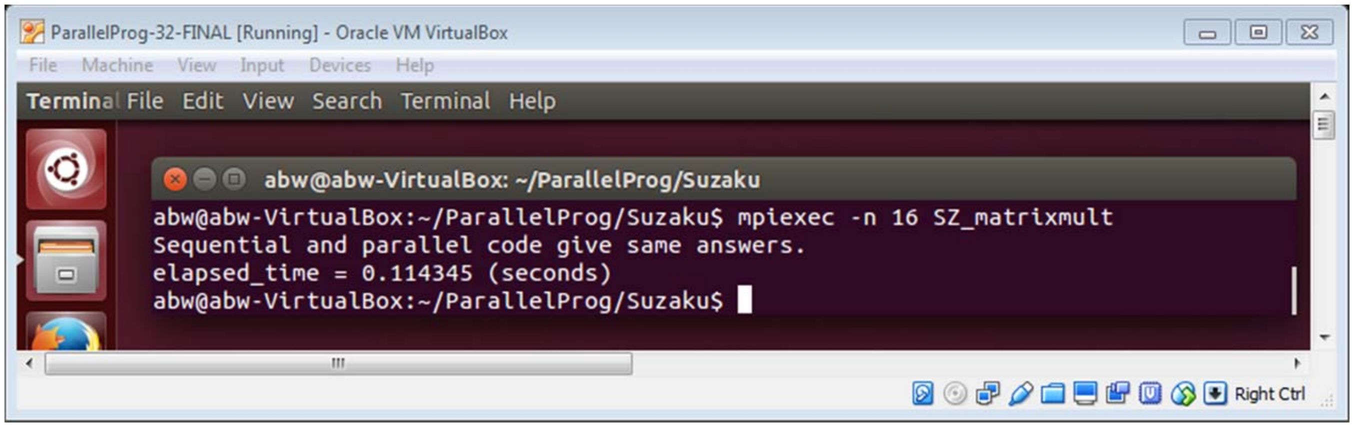
```
            }

            SZ_Gather(C1,C);        // gather results

        SZ_Parallel_end;         // end of parallel, back to just master, 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;
}
```

After SZ_parallel_end, only master process executed code.

Suzaku SZ_ matrixmult.c program

The matrices are initialized with values within the program rather than reading an input file. 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.

Sample output


ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Terminal File Edit View Search Terminal Help
abw@abw-VirtualBox: ~/ParallelProg/Suzaku
Right Ctrl

```
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpiexec -n 16 SZ_matrixmult
Sequential and parallel code give same answers.
elapsed_time = 0.114345 (seconds)
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

4. Nbody Program

Set as an assignment.