Suzaku Pattern Programming Framework Specification

Version 1.0

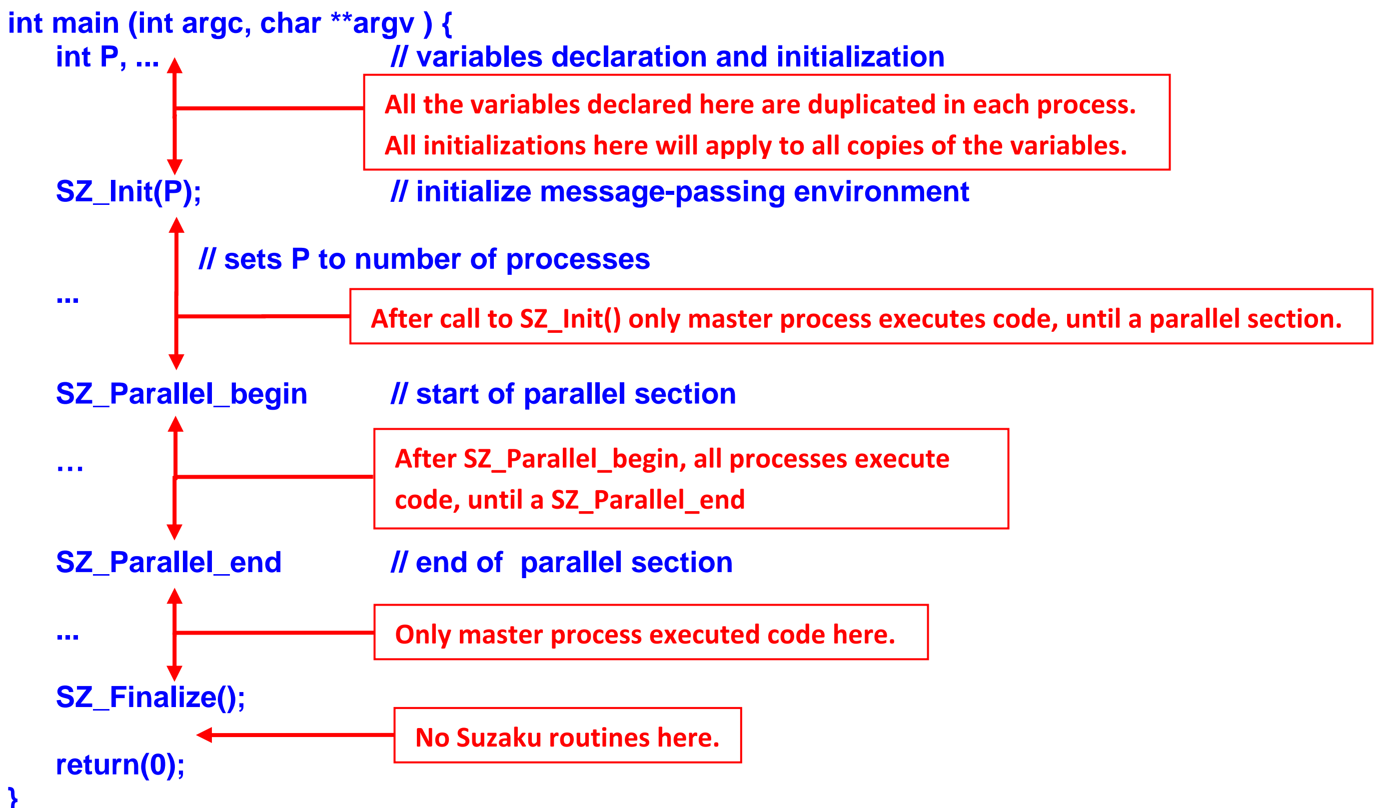
B. Wilkinson, March 14, 2016.

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.

1. 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.



Suzaku program structure

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:

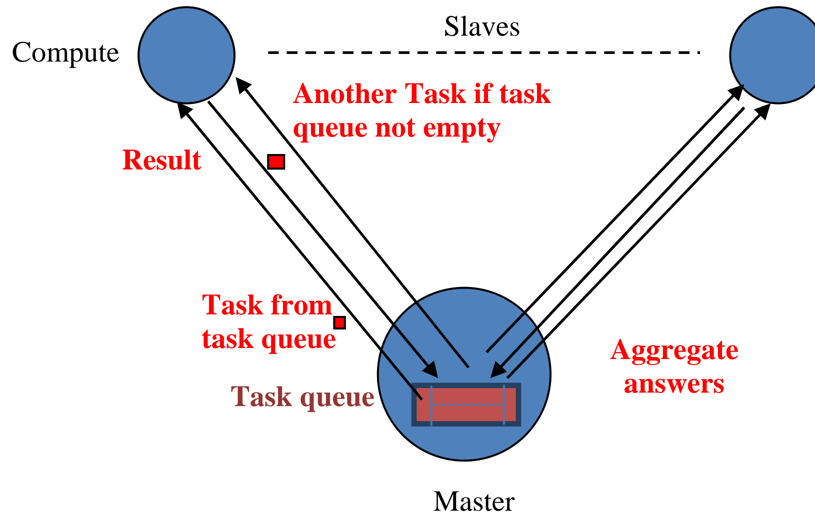
```
mpiexec -n <no_of processes> ./prog1
```

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

2. Workpool Pattern - Version 1

2.1 Workpool

The workpool pattern is like a master-slave pattern but has a task queue that provides load balancing, as shown below. 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



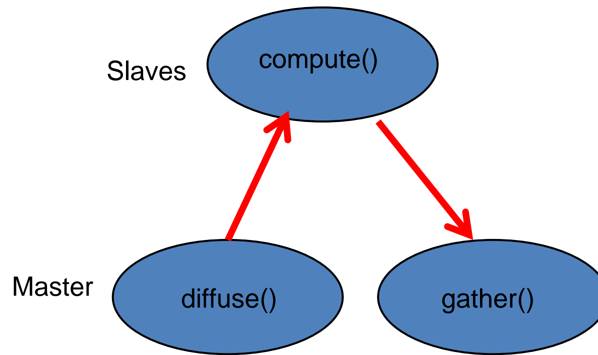
master waits until all outstanding results are returned. The termination condition is the task queue empty and all result collected.

Workpool with a task queue

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. 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 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 Seeds framework. 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
- **gather()** Accepts a slave result and uses it to develop the final answer



diffuse, compute, and gather routines

diffuse(), **compute**(), and **gather()** are dependent upon the application, and often very short.

Workpool code. The workpool itself is implemented by the provided routine **SZ_Workpool()** placed within a parallel section. **init**(), **diffuse**(), **compute**(), and **gather()** are called by **SZ_Workpool()** and given as input parameters. They can be re-named to accommodate for example multiple workpools in a single program.

2.2 Program structure

The program structure is shown below and consists of the four programmer 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]) {
        ...
        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;
}
```

Workpool program structure

2.3 Signatures of Programmer-Written Routines

Init

```
        void init(int *tasks, int * data_items, int *result_items)
```

This routine will be called at the beginning of the workpool by all processes. At the very least, it must set values for number of tasks (**T**), number of data items in each task (**D**), and number of data items in each result (**R**). The routine may be used for other initialization purposes. There is no implicit synchronization.

Parameters (pointers to integers):

int *tasks	Input parameter for the number of tasks
int *data_items	Input parameter for the number of data items (doubles) in each task
int *result_items	Input parameter for the number of data items (doubles) in result of each task

Limitations: The number of tasks, **T**, must be equal or less than INT_MAX – 1, but this is very unlikely to be an issue. For **D** and **R**, the limiting factor is the maximum size of dynamic arrays on the platform.

A typical coding sequence would be:

```
#define T 6                     // number of tasks, one task for each body
#define D 30                    // number of data items in each task
#define R 5                     // number of data items in result of each task

void init(int *tasks, int *data_items, int *result_items) { // all processes execute this at beginning
        *tasks = T;
        *data_items = D;
        *result_items = R;
        ...
        return;
}
```

Any names can be used in the formal parameter list.

Diffuse

The signature of this routine is:

```
void diffuse(int taskID, double output[D])
```

This routine generates the next task when called by the master.

Parameters:

int taskID	Input parameter for the task ID for the associated task
double output[D]	Output parameter for the task data, given as an array of **D** doubles

Notes: **taskID** is provided by the framework, from 0 to **T** - 1. Each time diffuse is called, **taskID** is incremented. **taskID** is carried with the task throughout the workpool. **taskID** provides a mechanism to do specific actions with particular tasks or results. When used by the programmer, **taskID** corresponds to a segment number in Seeds.

Compute

The signature of this routine is:

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

This routine is executed by a slave. It takes a task generated by diffuse and generates the corresponding result.

Parameters:

int taskID	Input parameter for the task ID for the associated task
double input[D]	Input parameter for the task data, given as an array of D doubles
double output[R]	Output parameter for the result, given as an array of R doubles

Gather

The signature of this routine is:

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

This routine accepts a slave result and develops the final answer. Called by the master.

Parameters:

int taskID	Input parameter for the task ID for the associated task
double input[R]	Input parameter for the slave result, given as an array of R doubles

Notes: **gather()** is used to aggregate the answer from the task results during the workpool operation, and programmers are free to do this any way they like and whatever the application dictates. To be able to reach the answer from outside gather, the final answer can be declared globally at the top of the program outside main.

2.4 Signature of Suzaku Workpool Routine

This routine is provided and implements the workpool. It calls **init()**, **diffuse()**, **compute()**, and **gather()**. **SZ_Workpool()** must be called within a Suzaku parallel section. The signature of the routine is:

```
void SZ_Workpool ( void (*init)(int *T, int *D, int *R),
                   void (*diffuse)(int *taskID,double output[]),
                   void (*compute)(int taskID, double input[], double output[]),
                   void (*gather)(int taskID, double input[])  )
```

Parameters:

***init**	Function pointer to init function
***diffuse**	Function pointer to diffuse function
***compute**	Function pointer to compute function
***compute**	Function pointer to gather function

Notes: The function pointers could have been eliminated if their names were standardized (as in Seeds), e.g. init, diffuse, compute, and gather, but specifying the function names makes it more obvious which functions are used by the workpool, and also allows multiple workpools each using different function pointers. No data arrays need to be declared for Suzaku. These are generated by Suzaku dynamically.

The programmer can implement routines outside the workpool as the need arises. Results from the gather need to be used to create the final answer and variables declared outside main to be reachable from gather and other routines.

2.5 Compilation and Execution

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

mpicc -c -o suzaku.o suzaku.c

to create an object file **suzaku.o** (note the -c option). This avoids having to recompile **suzaku.c** every time you compile application code.

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 an application workpool program **prog1.c**, issue the command:

mpicc -o prog1 prog1.c suzaku.o

A make file is provided for the sample programs.

Instead of pre-compiling **suzaku.c** into **suzaku.o**, one could also compile both **suzaku.c** and **prog1.c** together with:

mpicc -o prog1 prog1.c suzaku.c

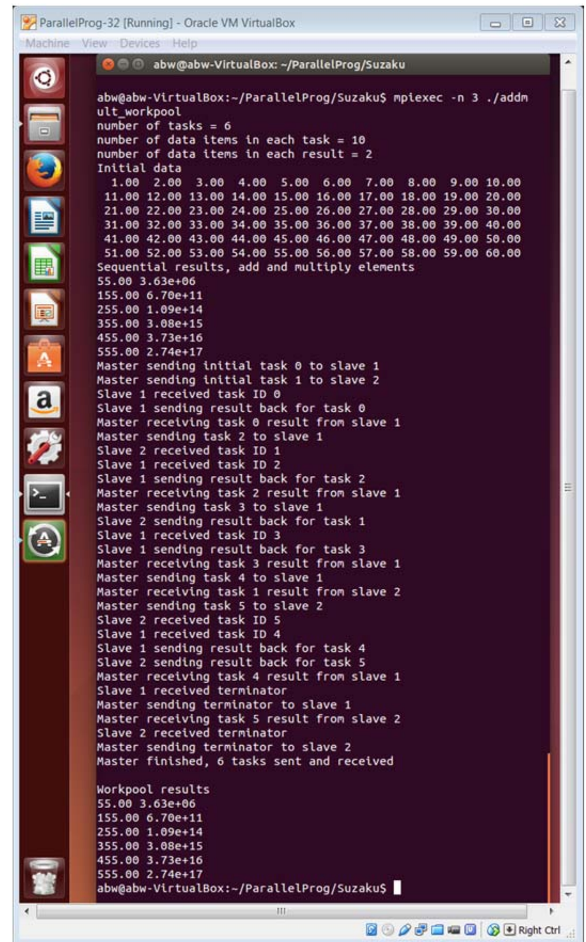
To execute **prog1**, issue the command:

mpiexec –n <no_of processes> prog1

where **<no_of processes>** is the number of processes you wish to use. 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 in Part 1 because collective routines are not used.

2.6 Debug 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**. To use, rename **SZ_Workpool()** in the application code to **SZ_Workpool_debug()** and recompile.



Sample output with debug messages

3. Workpool Version 2

A version of the Suzaku workpool has been implemented that mirrors the interface in Seeds by using "put" to pack data into tasks and results and "get" to retrieve the data. To differentiate between the versions, the initial version of the workpool is called version 1 and the workpool with put and get routines is called version 2. Version 2 may incur a greater overhead that version 1 but may be more elegant to use and is basis of the dynamic workpool described in Section 4.

Put Routine

The put routine is used by the programmer to insert data into a task and is called in the compute routine, once for each data item inserted into the task. The signature is:

SZ_Put(char[8] key, void *x)

Purpose: Places data into the send buffer and associates a user-defined name to it.

Parameters:

key String or string constant
x Pointer to data being stored in the message buffer and mapped to key

Limitations: **x** can be an individual character variable, integer variable, double variable or 1-dimensional array of characters, integers, or doubles, or a multi-dimensional array 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. **key** is a programmer selected string to identify the data, up to eight characters and there is a maximum of 10 keys (i.e. 10 puts to the same message buffer). These size limitations could be increased if needed but the mapping is attached to the message and so incurs an overhead.

Get Routine

The get routine is used by the programmer to extract data from a task and is called in the diffuse routine, once for each data item extracted from the task. The signature is:

SZ_Get(char[8] key, void *x)

Purpose: Extract data from the received message that is associated with a user-defined name.

Parameters:

key String or string constant
x Pointer to data being retrieved from the message buffer mapped to key

Limitations: **x** can be an individual character variable, integer variable, doublevariable, or 1-dimensional array of characters, integers, or doubles, or a multi-dimensional array 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. **key** is a string up to eight characters and there is a maximum of 10 keys (i.e. 10 puts to the same message buffer). These size

limitations could be increased if needed but the mapping is attached to the message and so incurs an overhead.

The workpool routines **init(), diffuse()**, **compute()**, and **gather()** now have different and simplified signatures:

init()

The **init()** routine now only has to set the number of tasks, **T**. **D**, the number of data items in each task and **R**, the number of data items in result of each task are not now used as they are determined with the put routines, i.e., the signature of **init()** is:

void init(int *T)

Parameter:

int *T	Input parameter for the number of tasks (pointers to an integer)

diffuse()

The **diffuse**() only needs the input parameter from the framework to provide the **taskID**. The output parameter **output[]** is not needed, i.e., the signature of **diffuse**() is:

void diffuse(int tasksID)

Parameter:

int taskID	Input parameter for the task ID for the associated task

compute()

The **compute()** only needs the input parameter from the framework to provide the **taskID**. The input parameter **input[]** and output parameter **output[]** are not needed, i.e., the signature of **diffuse()** is:

void diffuse(int tasksID)

Parameter:

int taskID	Input parameter for the task ID for the associated task

gather()

The **gather()** only needs the input parameter from the framework to provide the **taskID**. The input parameter **input[]** is not needed, i.e., the signature of **gather**() is:

void gather(int tasksID)

Parameter:

int taskID Input parameter for the task ID for the associated task

Suzaku Workpool Routine

The workpool routine is now called **SZ_workpool2** and has the signature:

```
void SZ_Workpool2 (void (*init)(int *T),
                   void (*diffuse)(int *taskID),
                   void (*compute)(int taskID),
                   void (*gather)(int taskID) )
```

Parameters:

***init**	Function pointer to init function
***diffuse**	Function pointer to diffuse function
***compute**	Function pointer to compute function
***compute**	Function pointer to gather function

Implementation Details

The put and get operations are achieved by using the MPI pack mechanism that enables a message to be constructed with multiple data items of any type and the MPI unpack mechanism that enables the data to be extracted from the packed message. To provide the greatest flexibility a lookup table is created that associates the key with the position in the buffer where the variable is packed, and this look-up table is attached to the complete message before the message is sent. Then **SZ_Get()** can be called in any order and also **SZ_Put()** can be called in any order. Also each message could have the same or different named data if required. This implementation does not need the input and output buffers to be declared by the programmer and are not parameters in diffuse, compute, and gather. The only parameter needed is the **taskID**. The size of the **x** is not needed, but **x** must be declared statically such that **sizeof()** can be used. Both **SZ_Put and SZ_Get** are macros in **suzaku.h** that find the size using **sizeof** and then each call a routine (**SZ_pack_data()** and **SZ_unpack_data()** respectively) in the workpool program in **suzaku.c**. These routines then call a routine to map the data to a key or unmap the data given a key before packing or unpacking. **SZ_Put** and **SZ_Get** do not return error values but routines they call can create error messages, notably if the allocated memory space is exhausted when mapping or packing, or the name cannot be found in the map in "unmapping."

The map is implemented as two 1-D arrays, one array holding the keys as character strings and the other holding the positions in the message buffer as integers. The map can be attached at the front or the end of the message. Both have been tried. At the front requires a fixed sized map for all messages so that the start of the data is known before mapping. At the end requires a pointer to it at the front and potentially the map could be a different size for each message although that was not implemented.

If the programmer uses **SZ_Put()** or **SZ_Get()** within control statements such as if statements, it is safest to include braces, e.g.

```
if (task_no == 0) { SZ_Put("mydata",data1); }  else {SZ_Put("mydata",data2);}
```

because macros do in-line substitution and consist of multiple statements. (Internally they are wrapped around do { … } while(0); statements but that is not sufficient.)

Also in the example given, all messages sent will have data called “mydata.” If one sends messages with different named data, one would need to recognize that at the destination. Using **SZ_Get()** with a name that does not exist in the message will result in an error message (“name not found”).

Compilation and Execution

SZ_Workpool2() and associated routines are held in **suzaku.c**. Compilation and execution is the same as for workpool version 1 except for naming the workpool as **SZ_Workpool2()** in the application code.

4. Workpool Version 3

This version of the workpool implements a workpool where new tasks can be added to the task queue during the computation as might be needed for problems such as the shortest path problem. We call this a dynamic workpool as opposed to the static workpool where the number of tasks in the task queue is fixed.

The routines **SZ_Put()** and **SZ_Get()** are available from version 2 to add data to task and results. In addition one new routine, **SZ_Insert_task()**, is available for use by the programmer to add tasks to the task queue:

SZ_Insert_task

The signature of this routine is:

```
int SZ_Insert_task(int taskID)
```

Purpose: This routine adds a task to the task queue.

Parameter:

int taskID Input parameter for the task ID for the associated task, provided by the framwork

Return value: An integer giving the number of tasks in the task queue afterwards or -1 if the tasks queue is already full and a task cannot be added.

Limitations: The task queue is maintained by the master process and not accessible by the slaves. Hence the routine can only be called by the master process, either within **init**(), **diffuse(),** or **gather()**, i.e. it cannot be used by the slaves in **compute**().

Note: The programmer is not expected to remove tasks for the task queue as this will be done by the framework.

Suzaku Workpool Version 3 Routine

Version 3 is based upon version 2 and purposely uses the same signature as version 2 (except the workpool routine name, **SZ_Workpool3**()):

```
void SZ_Workpool3 (void (*init)(int *T),
                   void (*diffuse)(int *taskID),
                   void (*compute)(int taskID),
                   void (*gather)(int taskID)  )
```

Parameters:

***init**	Function pointer to init function
***diffuse**	Function pointer to diffuse function
***compute**	Function pointer to compute function

***compute** Function pointer to gather function

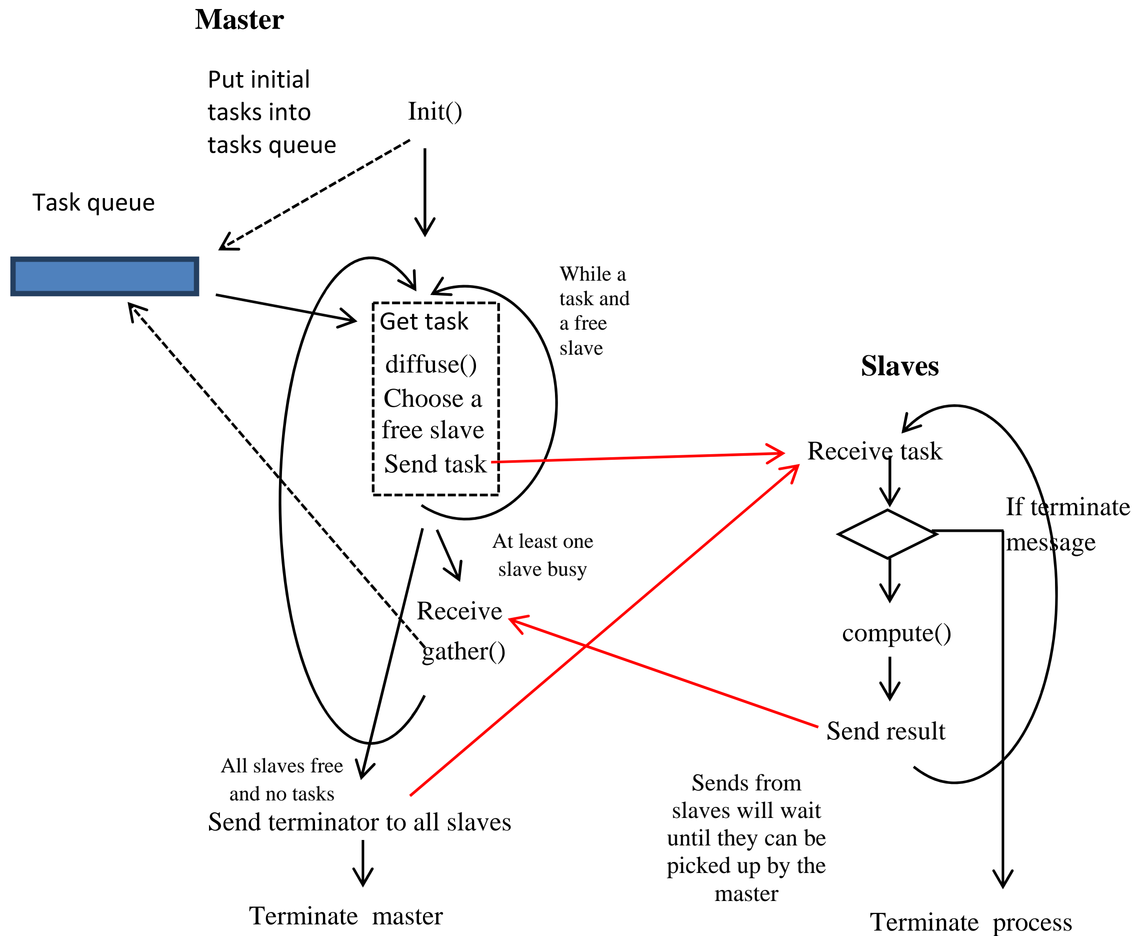
init() now needs to initialize the task queue using **SZ_Insert_task()** instead of specify the number of tasks but for compatibility with version 2, the input parameter ***T** (no of tasks) is retained. If the number of tasks is set to a number greater than 0, version 3 will implement the static workpool by automatically initializing the task queue for one task (**taskID** = 0) and inserting a consecutive task when a task is taken from the queue, up to **T** tasks.

Compilation and Execution

SZ_Workpool3() and associated routines are held in **suzaku.c**. Compilation and execution is the same as for workpool version 2 except for naming the workpool as **SZ_Workpool3()** in the application code. *Version 3 can be used with version 2 application programs without any other change to the application code.*

Implementation Details

The workpool algorithm implemented for Version 3 is shown below:



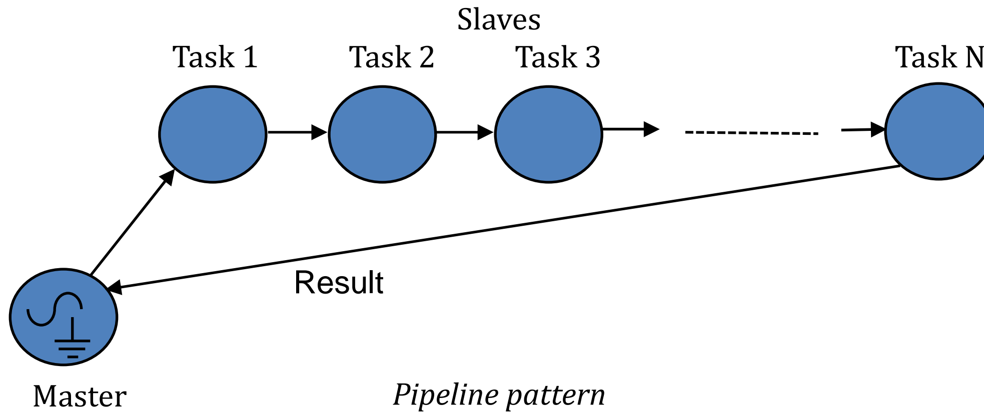
The task queue is a first-in first-out queue. Tasks are identified by an integer taskID, which could be duplicated and are not necessarily unique consecutive numbers as in version 2. (If a particular instance

of a task needs to be differentiated further, that information can be added by the programmer, see later.) It is also necessary to maintain information about the slaves. Whenever a message is sent to a slave, slave set as busy and number of busy slaves incremented by 1. Whenever a result is received back slave set as free and number slaves decremented by 1. A slave has to be chosen from those free and a round-robin algorithm is used.

Initially the task queue is initialized with at least one task in the workpool routine **init()**. Then a task is retrieved from the task queue, **diffuse()** is executed and the complete task with any addition information added by diffuse is sent to a free slave if there is one. When there are no more free slaves or no more tasks, the master process waits for one slave to return a result. Slaves accept tasks, execute **compute()**, and return results, which could include new tasks packed into an integer array. The master picks up the results of one slave, and executes a **gather()** routine provided with the task ID. The gather routine might find new tasks to add to the task queue. The master then repeats the complete sequence taking tasks from the task queue and sending tasks to free slaves, etc. The sequence stops when there are no new tasks and all slaves are free. Then all slaves are terminated with termination messages from the master and the master terminates. This algorithm avoids needing to use concurrent processes or threads for diffuse and gather, which were tried but is complicated by the need for shared memory, critical sections, and an MPI implementation that is thread-safe for the thread-based solution.

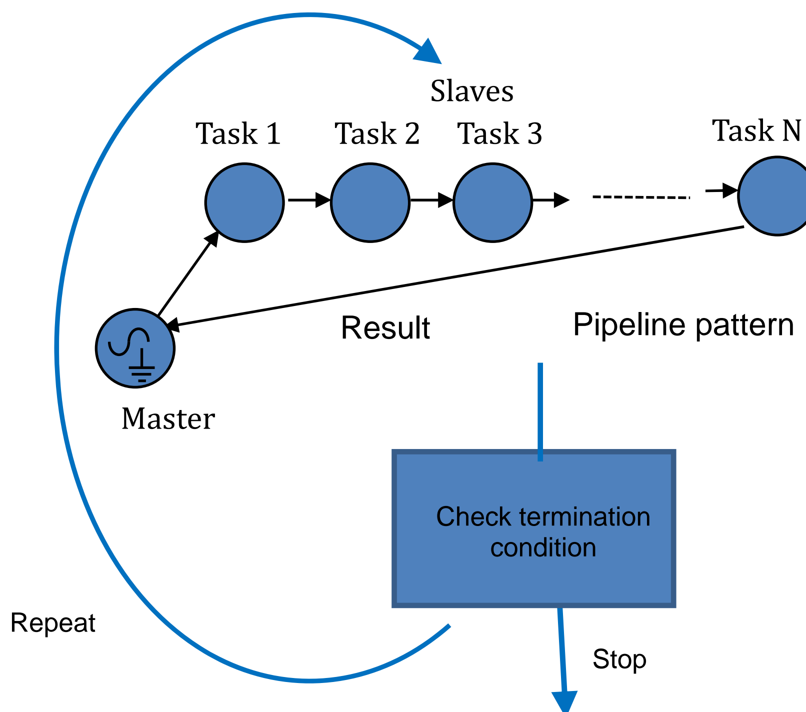
5. Pipeline Pattern

The iterative synchronous pipeline pattern has been implemented. In the pipeline pattern, the computation is divided into a series of tasks that have to be performed one after the other, with the result of one task passed on to the next task, like an assembly manufacturing line. One computational unit, a slave here, performs each task:



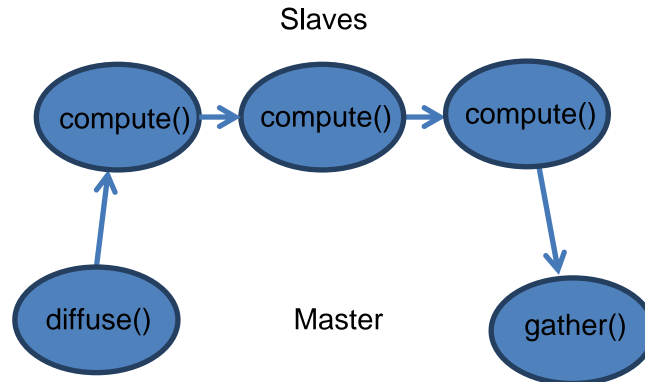
Pipeline pattern

In the iterative synchronous pipeline pattern, the pipeline is within an iteration loop, to achieve increased performance as in an assembly line. At each iteration, tasks pass from one process to the adjacent process in the pipeline.



Suzaku Pipeline

The programmer's interface is purposely similar to other patterns. The slaves execute the compute routine and the master executes the diffuse and gather routines:



This approach is the same as the pipeline pattern in Seeds. The basic version uses 1-D arrays as in the workpool version 1 as this is the most likely data structure and most efficient implementation although there is no technical reason why version 2 put and get mechanism could not be incorporated.

The programmer must implement:

- **void init()** To initialize the number of tasks, **T**, and the size of each task, **D** at beginning of computation. Executed by all processes. Other initializations can be done
- **void diffuse ()** Generates next task when called by master. Sent to the first slave
- **void compute()** Executed by the slaves. Takes task received and generates corresponding result
- **void gather()** Accepts result from final slave and develops final result. Called by Master.

Signatures:

The signatures art the same as the workpool version 1:

void diffuse (int taskID, double output[N])
void compute (int taskID, double input[N], double output[N])
void gather (int taskID, double input[N])

The pattern is implemented by **SZ_Pipeline()** with the signature:

```
SZ_Pipeline( void (*init)(int *T, int *D, int *R),
             void (*diffuse)(int *taskID,double output[]),
             void (*compute)(int taskID, double input[], double output[]),
             void (*gather)(int taskID, double input[])  )
```

Termination

The pipeline will terminate naturally after T * (P-1) steps where are T tasks and P processes. A routine is provided to be able to terminate the pattern earlier when a termination condition exists:

```
void SZ_Terminate()
```

This routine would be called by the gather routine.

Debugging

A routine is provided that will cause debug messages to be displayed during the pipeline operation:

```
void SZ_Debug()
```

This routine would be placed immediately before **SZ_Pipeline**() with parallel section. This approach could be used in others rather than provide two separate routines as in the workpool version 1. (Not sure which is best yet.) With pre-implemented patterns it is really important to be able to understand and watch the execution steps as the programmer does not have access to the underlying implementation.

Program structure

The program structure is similar to a workpool and shown below, consisting of the four programmer 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]) {
        ...
        return;
}

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

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

```
    SZ_Parallel_begin

        SZ_Debug();

        SZ_Pipeline(init, diffuse, compute, gather);

    SZ_Parallel_end;

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

    SZ_Finalize();

    return 0;
}
```

Pipeline program structure

Compilation and Execution

SZ_Pipeline() and associated routines are held in **suzaku.c**. Compilation and execution is the same as for other Suzaku patterns.

6. Generalized Patterns

Message passing patterns connect sources and destinations together in various ways. The master-slave pattern connects the master to slave processes but the slaves are not interconnected. Any communication between slaves has to go through the master. The master-slave pattern is the basic pattern for parallel programming using a computational strategy of dividing the work into parts to be done by the slaves. The (static) workpool extends the master-slave pattern to include a task queue providing a load balancing feature. The dynamic workpool extends it further to enable new tasks to be added during the communication. For particular algorithms, a specific interconnection pattern might offer advantages and the pipeline pattern is one such specialized pattern. Other such patterns include the stencil pattern in which slaves are arranged in two or three dimensional meshes and each slave connects to its neighbors in the mesh. The stencil pattern can be generalized into what we call *overlapping connectivity* patterns where how near a neighbor should be to be connected can be different, leading to many nearest neighbor patterns. The extreme case is where each slave connects to all the other slaves in an *all-to-all pattern*. The other extreme is where each slave connects to only one other slave. The (unidirectional) pipeline pattern connects each slave to the next slave in the pipeline. There are many other possible connection patterns for example binary trees and arbitrary connection patterns for specific problems.

Rather than implement every pattern in a unique way, the approach taken in Suzaku is to implement a pattern based upon a directed graph called here a *connection graph*. We call this approach a *generalized pattern*. Any connection pattern can be created this way. Of course one has to avoid messaging deadlock in the pattern implementation and it may be the implementation is not as efficient as specific implementations for specific patterns.

Most patterns are repeated as synchronous iterative patterns terminating when a termination condition occurs, usually either a fixed number of iterations or when the computed values converge sufficiently (i.e. do not change by more than a given value). The Suzaku generalized pattern is a synchronous iterative pattern with a master-slave structure as illustrated in Figure 1. The master sends initial data to all slaves and collects results from all slaves at the end of the computation. The slaves compute and send values to those slaves that are interconnected, repeatedly until the termination condition exists. *The master also acts as one slave as in the master-slave pattern.* For greatest flexibility, the programmer implements the iteration loop and low level routines are provided that the programmer to construct the pattern. Broadcast/scatter/gather rely on using existing low level Suzaku routines. Two additional routines are provided for the programmer:

SZ_Pattern_init("pattern_name",T,D,R) – to initialize the connection graph for standard patterns (all-to-all, pipeline, stencil so far)

SZ_Pattern_send(A,B) – to send an array from each slave to all connected slaves according to the connection graph

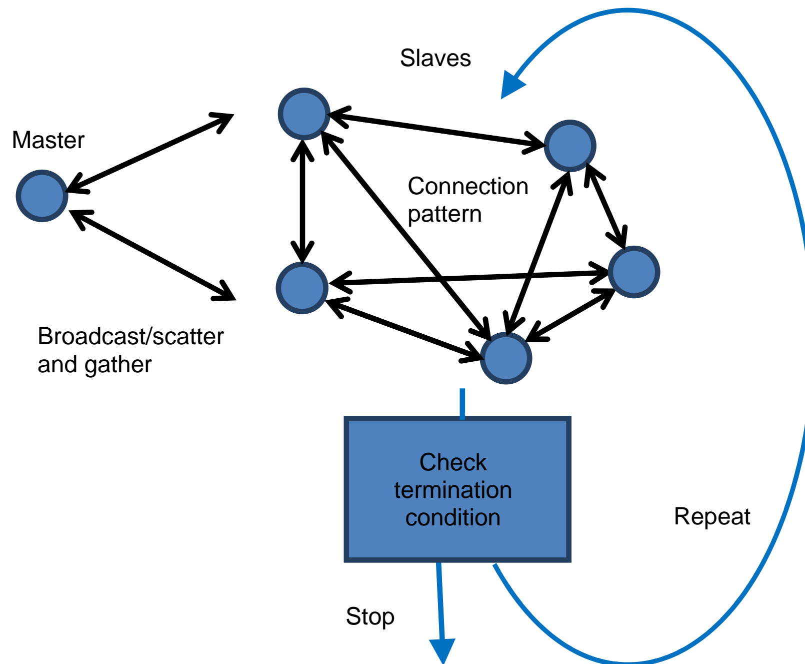


Figure 1 Generalized iterative synchronous pattern

Overall structure

```
SZ_Parallel_begin                          // parallel section, all processes do this

    SZ_Pattern_init("pattern_name",T,D,R);  // set up slave interconnections in each slave
     …                                      // initialize data, input and output arrays

    SZ_Broadcast(input);                    // broadcast initial data to all slaves, if needed

    for (i = 0; i < steps; i++) {           //  in this case a fixed number of iteration
        compute(i,input,output);            // slaves execute compute, master acts as a slave
        SZ_Pattern_send(output,input);      // sent compute results to connected slaves
     }

     SZ_Gather(input,result);               // collect results from slaves

SZ_Parallel_end;                           // end of parallel
```

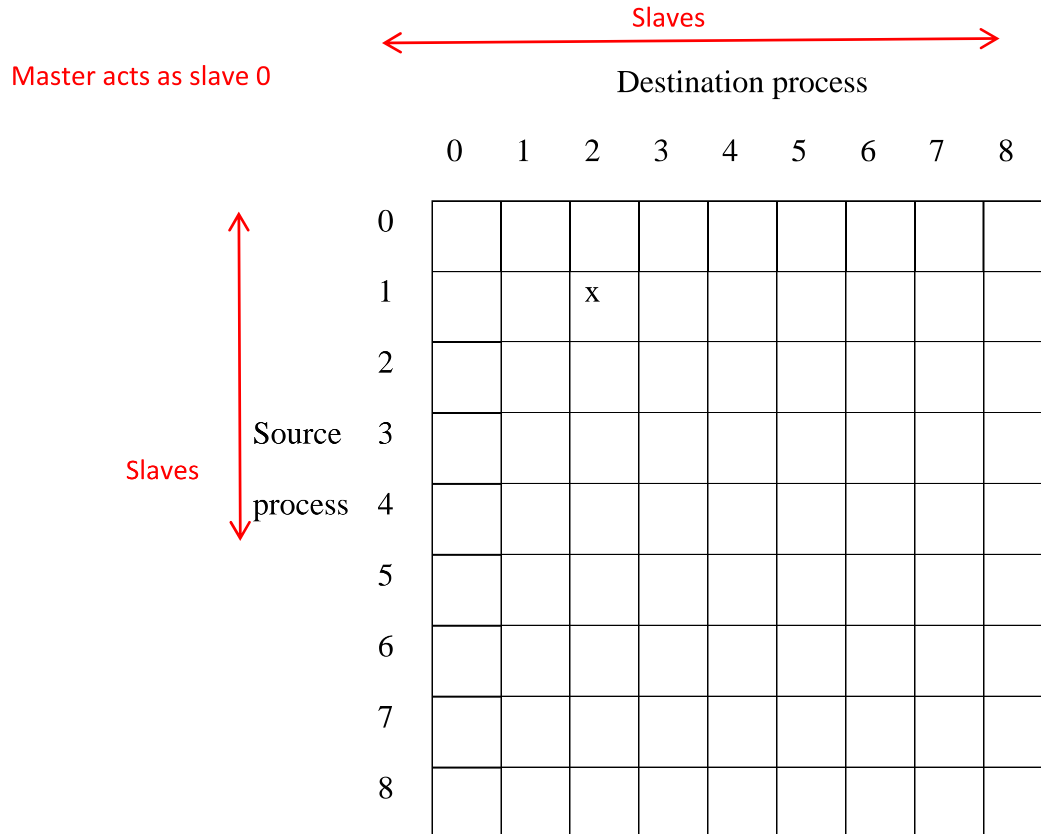
Each slave maintains two arrays **input[][]** and **output[]**. **Input[][]** holds the data sent from connected slaves. Slaves create results in **output[]** to be send to connected slaves. The connection graph specifies how the data is arranged in these arrays, see next.

The broadcast corresponds to *diffuse* in the general case and could be coded inside a routine called diffuse(). Similarly gather corresponds to *gather* in the general case and could be coded inside a routine called gather().

Implementation

To be able to use the provided routines, one needs to appreciate the connection graph.

Connection graph: The connection graph specifies which slaves are interconnected and the location in the destination for the incoming data. It is a directed graph so it would be possible for the connection to be in one direction although that would be unusual. The graph is a P x P adjacency matrix, **connection_graph[P][P]**, where there are P processes (slaves, as the master acts as one slave) and illustrated below for eight slaves:



The source data being transferred is a 1-D array of doubles, **output[N]**. The destination array is a 2-D array, **input[P][N]**. The graph entry at **connection_graph[i][j]** indicates:

- -1 No connection
- x A connection process *i* to process *j*, and the value, x, indicates the row in the destination array where the data is to be placed, i.e. **input[x][N]**.

This would allow a fully connected graph with each value received held in a separate location. In a partially connected graph, not all rows in **input[][]** would be used. The graph can be set up to create any pattern including all-to-all, pipeline, stencil, binary tree etc. The all-to-all, pipeline, and stencil patterns have been created so far. Once the pattern is created, a generalized send routine uses the connection graph and send **output[]** to all connected processes, storing the data in the designated row of **input[][]**.

The basic version sends 1-D arrays as in the workpool version 1 as this is the most likely data structure and most efficient implementation although there is no technical reason why version 2 put and get mechanism could not be used. The arrays hold doubles.

Sample connection graph patterns

1. "all-to-all"

Destination process

Source process	0	1	2	3	4	5	6	7	8
0	-1	0	0	0	0	0	0	0	0
1	1	-1	1	1	1	1	1	1	1
2	2	2	-1	2	2	2	2	2	2
3	3	3	3	-1	3	3	3	3	3
4	4	4	4	4	-1	4	4	4	4
5	5	5	5	5	5	-1	5	5	5
6	6	6	6	6	6	6	-1	6	6
7	7	7	7	7	7	7	7	-1	7
8	8	8	8	8	8	8	8	8	-1

x = source process ID

i.e. the array **output[N]** from slave *i* will be sent to the *i*th row of **input (input[i][N]**) and all locations of **input** will be used.

2. **" pipeline"** (note: a ring)

Destination process

Source process	0	1	2	3	4	5	6	7	8
0	-1	0	-1	-1	-1	-1	-1	-1	-1
1	-1	-1	0	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	0	-1	-1	-1	-1	-1
3	-1	-1	-1	-1	0	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	0	-1	-1	-1
5	-1	-1	-1	-1	-1	-1	0	-1	-1
6	-1	-1	-1	-1	-1	-1	-1	0	-1
7	-1	-1	-1	-1	-1	-1	-1	-1	0
8	0	-1	-1	-1	-1	-1	-1	-1	-1

x = 0

i.e. the array **output[N]** from slave *i* will be sent to the first row of **input (input[0][N]**) and **input[1][N]** … **input[N-1][N]** will not be used.

3. 2-D "stencil"

Slaves are arranged in a square 2-D mesh. The number of slaves must have an integer squareroot. Nine slaves gives a 3 x 3 stencil. Processes are numbered in natural order:

0 1 2
3 4 5
6 7 8

Apart from slaves at the edges, each slave connects to the four neighbors on left, right, up and down, e.g. process 4 connect to 1, 3, 5 and 7. The edges only connect to those slaves that exist, e.g. process 1 connects to 0, 2, and 4. In most stencil computations, a constant boundary value is used by the process in the computation where it does not have neighboring process.

Processes will receive up to four **output[]** arrays, one from each neighbor loaded into **input[0][]**, **input[1][]**, **input[2][]**, and **input[3][]**. Values for x:

From the process to the left	x = 0
From the process to the right	x = 1
From the process above it	x = 2
From the process below it	x = 3

to all each to be placed in different location.

Below is shown for a 3 x 3 stencil (9 x 9 connection graph):

Destination process

Source process		0	1	2	3	4	5	6	7	8
	0	-1	0	-1	2	-1	-1	-1	-1	-1
	1	1	-1	0	-1	2	-1	-1	-1	-1
	2	-1	1	-1	-1	-1	2	-1	-1	-1
	3	3	-1	-1	-1	0	-1	2	-1	-1
	4	-1	3	-1	1	-1	0	-1	2	-1
	5	-1	-1	3	-1	1	-1	-1	-1	2
	6	-1	-1	-1	3	-1	-1	-1	0	-1
	7	-1	-1	-1	-1	3	-1	1	-1	0
	8	-1	-1	-1	-1	-1	3	-1	1	-1

Routines (in suzaku.c)

SZ_Pattern_init

void SZ_Pattern_init(const char* pattern, int N)

Purpose: To initialize the connection graph , **connection_graph[P][P]** to one of various selectable patterns and create message buffer space for the generalized send routines. This routine provides a copy of the connection graph and message buffer space to all processes and is called within a parallel section before using the pattern with **SZ_Generalized_send()**.

Parameters:

pattern Name of the pattern as a string constant (input parameter). So far:
"all-to-all"
"pipeline" or “ring”

"stencil

N Number of data items, i.e. size of **output[]** (input parameter).

Limitation: **connection_graph[P][P]** is statically declared as 20 x 20 elements, setting the maximum number of slaves (processes) to be 20. It is not expected that **P** would be very large in most systems, but can be altered in **suzaku.c**. The actual size of **P** being used is established with **SZ_Init()**. The size **N** is not so limited as the message buffer is declared dynamically in **suzaku.c**. This routine must only be called within a parallel section.

SZ_Generalized_send

void SZ_Generalized_send(double *output, double *input)

Purpose: To send the array **output[N]** to all connected processes as specified in the connection graph. The destination process stores the array in row of **input[P][N]** given by the connection graph.

Parameters:

***output**	Pointer to the array **output[N]** in source process
***input**	Pointer to the array **input[P][N]** in destination

Limitation: It is assumed that **N** is the value set in **SZ_Pattern_init()** and **P** is the value set in **SZ_Init()** for indexing into the array **output[][]**. This routine must only be called within a parallel section.

User defined patterns

void SZ_Set_connection_graph(int *g)

Purpose: To set the **connection_graph[P][P]**to the values given by the user-supplied input array, g[][].

Parameter:

***g** Pointer to the array **g[P][P]** holding the pattern where P is the current number of processes

Debugging

void SZ_Print_connection_graph(void)

Purpose: To cause the master to print the current connection graph, **connection_graph[P][P]**, for test purposes.

Parameters: None

Implementation Notes: Messaging is done point to point and a barrier in present at the end to ensure all processes complete before returning, i.e. the routine is synchronous as are low level Suzaku message passing routines but it is implemented as a routine and not as a macro. If all **MPI_send()**'s precede **MPI_recv()**'s in a process, there is a possible deadlock if sends become synchronous because of lack of buffer storage. To avoid possible deadlock, the implementation uses MPI buffered sends with explicit buffer space. Beforehand calling **MPI_BSend()**for the first time in a process, it is necessary to call **MPI_Buffer_attach()** to attach a buffer. The size of the buffer needs to be only big enough for all pending sends in a process. Here each process just needs space for one message. At end of all sends **MPI_Buffer_detach()** should be called. **SZ_Pattern_finalize()** will do this if the programmer wants to use it.

Compilation and Execution

The generalized pattern routines are found in **suzaku.c**. Application code using them must be compiled with the math libraries, **-lm** option even if **suzuku.o** is recompiled.

Appendix A Sample Programs with Suzaku routines

1. Low level routines

Point-to-Point Pattern

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

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

int main(int argc, char *argv[]) {
                                            // All variables declared here are in every process
    int i, P, PID;
    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];
    char b_message[20];
    strcpy(a_message, "Hello, world");
    strcpy(b_message, "------------");
    double p;
    double q;
    p = 123;
```

Note: 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 MPI message-passing environment
                                            // sets P to number of processes, not used here
```

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, 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 an single number from one process to another

        if (PID == 1)  printf("Received by process %d = %f\n",PID,q); // print it out at destination

    SZ_Parallel_end;                        // end of parallel, implicit barrier

    SZ_Finalize();
    return 0;
}
```

Only master process executed code here.

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

Matrix Multiplication with Master-Slave Pattern

A sample program called **matrixmult.c** is given below that demonstrates many of the Suzaku macros.

```
#define N 64
#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 A1[N][N];        // used in slaves to hold scattered a
        double C1[N][N];        // used in slaves to hold their result
        double time1, time2;    // use clock 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

        SZ_Scatter(A,A1);       // Scatter A array into A1
        SZ_Broadcast(B);        // broadcast B array

        blksz = N/P;
        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;
                }
        }
```

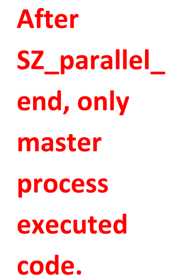
All the variables declared here are duplicated in each process. All initializations here will apply to all copies of the variables.

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);

        SZ_Finalize();
        return 0;
}
```



Suzaku 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.

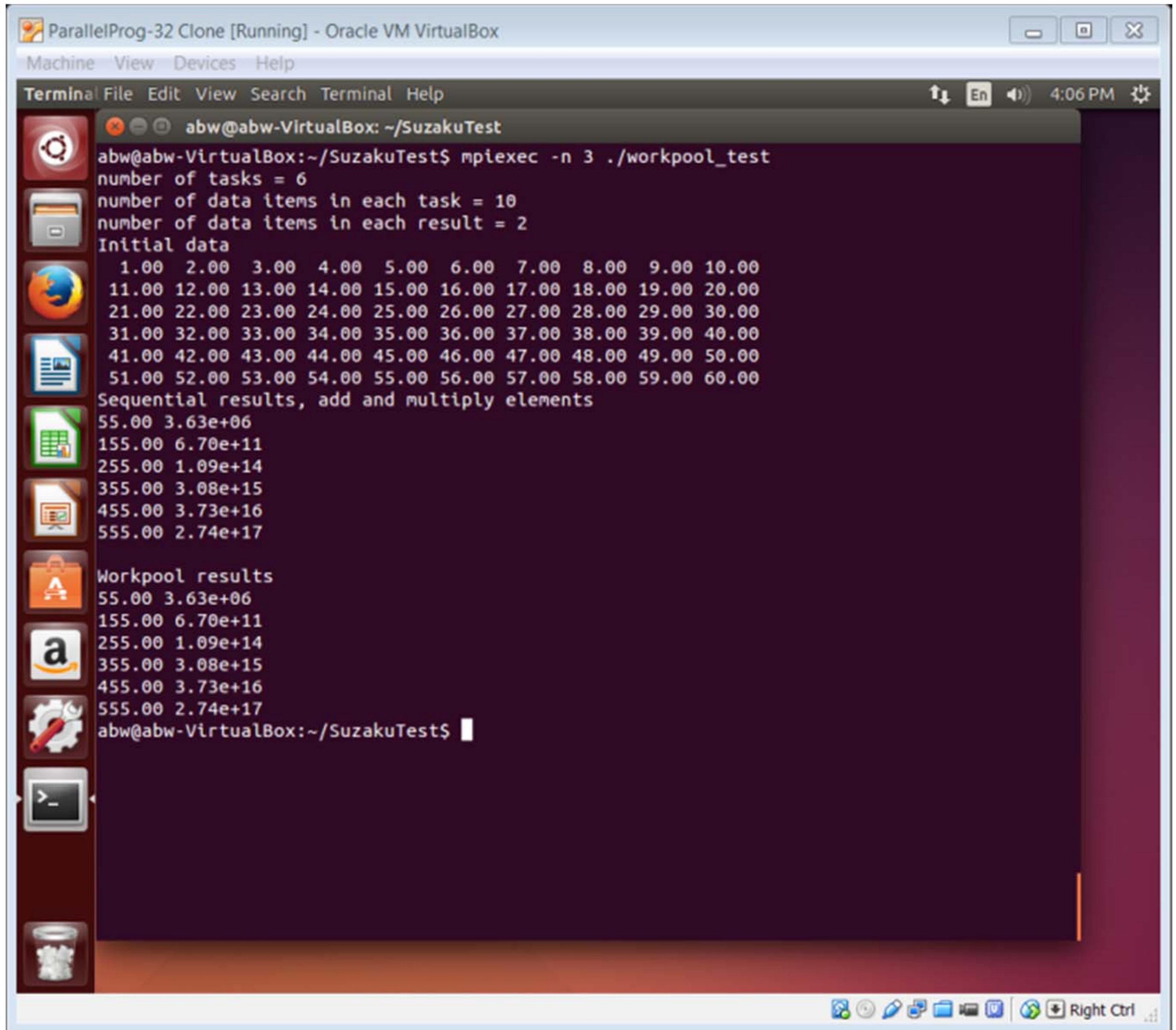
2 Workpool Version 1

Adding and Multiplying Numbers

```
// Suzaku Workpool pattern version 1 Application:  Adding and multiplying numbers. B. Wilkinson April 3, 2015
#include <stdio.h>
#include <string.h>
#include "suzaku.h"

#define T 6			// number of tasks, max = INT_MAX - 1
#define D 10			// number of data items in each task, doubles only
#define R 2			// number of data items in result of each task, doubles only

// No arrays need to be declared for Suzaku. Following used in this particular application, not required in general

double workpool_result[T][R];		// Final results computed by workpool, in gather()
double task[T][D];			// Initial data, T tasks, each task D elements, created by initialize() for
testing

// workpool functions to be provided by programmer:

void init(int *tasks, int *data_items, int *result_items) {
	*tasks = T;
	*data_items = D;
	*result_items = R;
	return;
}

void diffuse(int *taskID,double output[D]) {
	int j;				// taskID not used
	static int temp = 0;			// only initialized first time function called
	for (j = 0; j < D; j++)
		output[j] = ++temp;		// set elements to consecutive data values
}

void compute(int taskID, double input[D], double output[R]) {
				// function done by slaves -- simply adding the numbers together, and multiply them
	output[0] = 0;
	output[1] = 1;
	int i;
	for (i = 0; i < D; i++) {
		output[0] += input[i];
		output[1] *= input[i];
	}
	return;
}

void gather(int taskID, double input[R]) {		// function done by master collecting slave results
					// Final results computed by master, uses taskID
	int j;
	for (j = 0; j < R; j++) {
		workpool_result[taskID][j] = input[j];
	}
}

// additional routines used in this application

void initialize() {			// create initial data for sequential testing, not used by workpool

	int i,j;
	int temp = 0;
	for (i = 0; i < T; i++) {		// initialize data
		for (j = 0; j < D; j++)		// set elements to consecutive data values
			task[i][j] = ++temp;
```

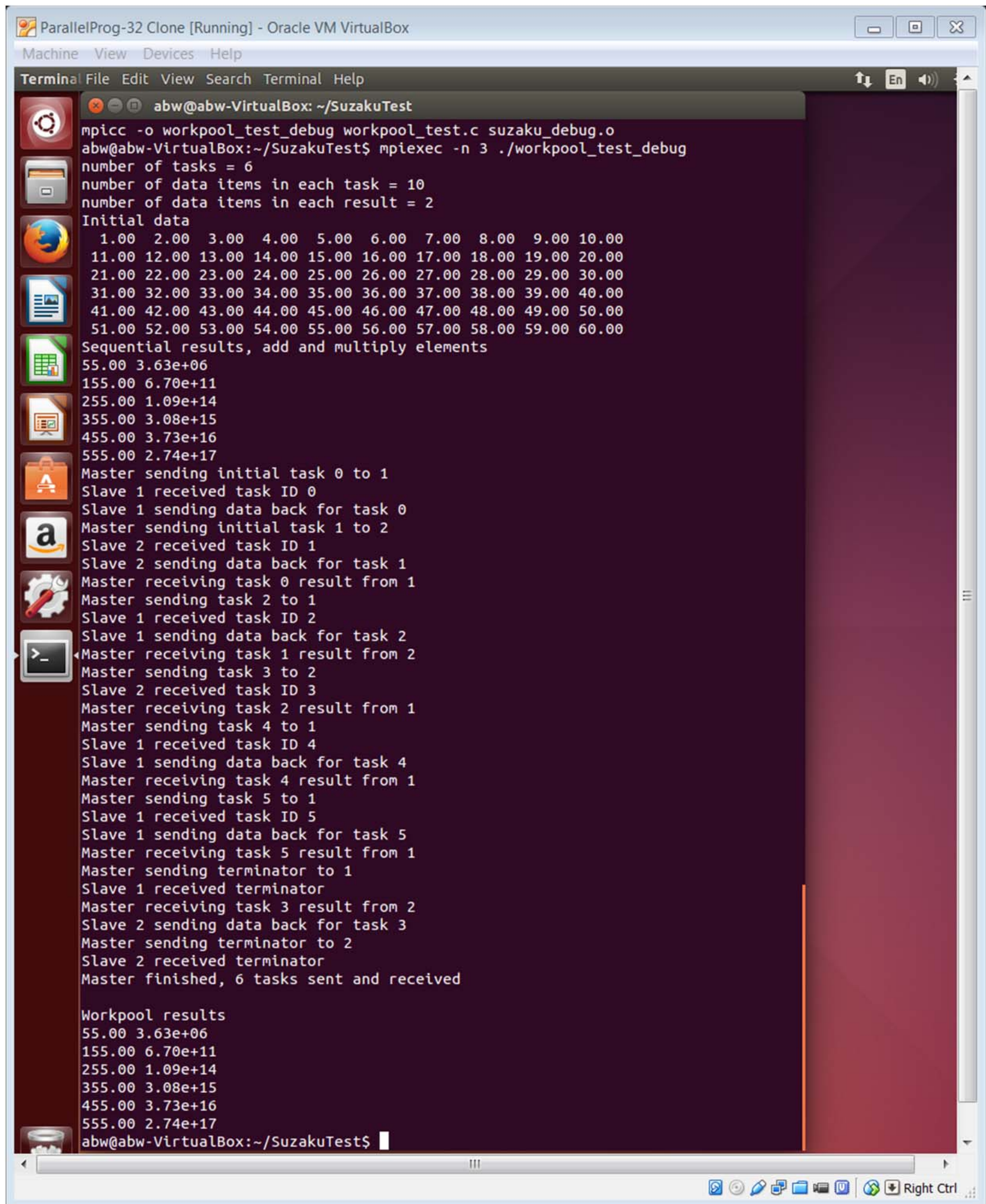
```
		}
		printf("Initial data\n");				// print out data
		for (i = 0; i < T; i++) {
			for (j = 0; j < D; j++)
				printf(" %5.2f",task[i][j]);
			printf("\n");
		}
}

void compute_seq() {					// Compute results sequentially and print out

		int i,j;
		double seq_result[T][D];		//** Final results computed sequentially
		printf("Sequential results, add and multiply elements\n");  // print out results
		for (i = 0; i < T; i++) {
			seq_result[i][0] = 0;
			seq_result[i][1] = 1;
			for (j = 0; j < D; j++) {
				seq_result[i][0] += task[i][j]; // add up all numbers in task result in [0]
				seq_result[i][1] *= task[i][j]; // multiply up all numbers in task result in [1]
			}
			printf("%5.2f ", seq_result[i][0]);		// print result 0
			printf("%5.2e", seq_result[i][1]);		// print result 1
			printf("\n");
		}
}

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

		SZ_Init(P);				// initialize MPI message-passing, sets P to be number of
processes

		printf("number of tasks = %d\n",T);
		printf("number of data items in each task = %d\n",D);
		printf("number of data items in each result = %d\n",R);

		initialize();				// create initial data for sequential testing, not used by workpool
		compute_seq();				// compute results sequentially and print out

		SZ_Parallel_begin

		SZ_Workpool(init,diffuse,compute,gather);

		SZ_Parallel_end;			// end of parallel

		printf("\nWorkpool results\n");		// print out workpool results
		for (i = 0; i < T; i++) {
			printf("%5.2f ", workpool_result[i][0]);		// result
			printf("%5.2e", workpool_result[i][1]);		// result
			printf("\n");
		}
		SZ_Finalize();
		return 0;
}
```

Sample output



Version with debug messages

Sample output:



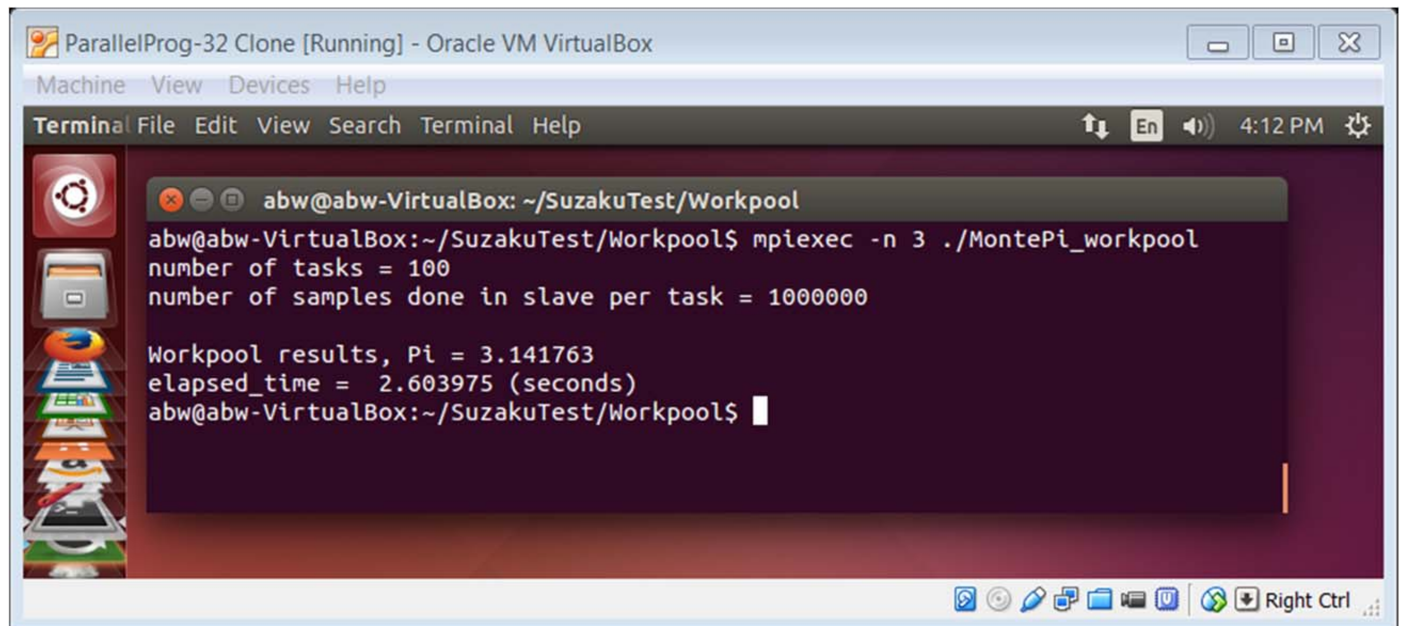
This version could be used for educational purposes.

Monte Carlo π calculation

```
// Suzaku Workpool pattern version 1 -- Application:  Monte Carlo Pi.  B. Wilkinson April 4, 2015

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.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

// constant used in computation
#define S 1000000               // sample pts done in a slave

// gobal variable
double total = 0;               // final result

// required workpool functions

void init(int *tasks, int *data_items, int *result_items) {
        *tasks = T;
        *data_items = D;
        *result_items = R;
        return;
}

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;
        return;
}

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

        total += input[0];              // aggregate answer
}

// 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[]) {
                                        // All variables declared here are in every process
        int i;
        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;
}
```

Sample output



–

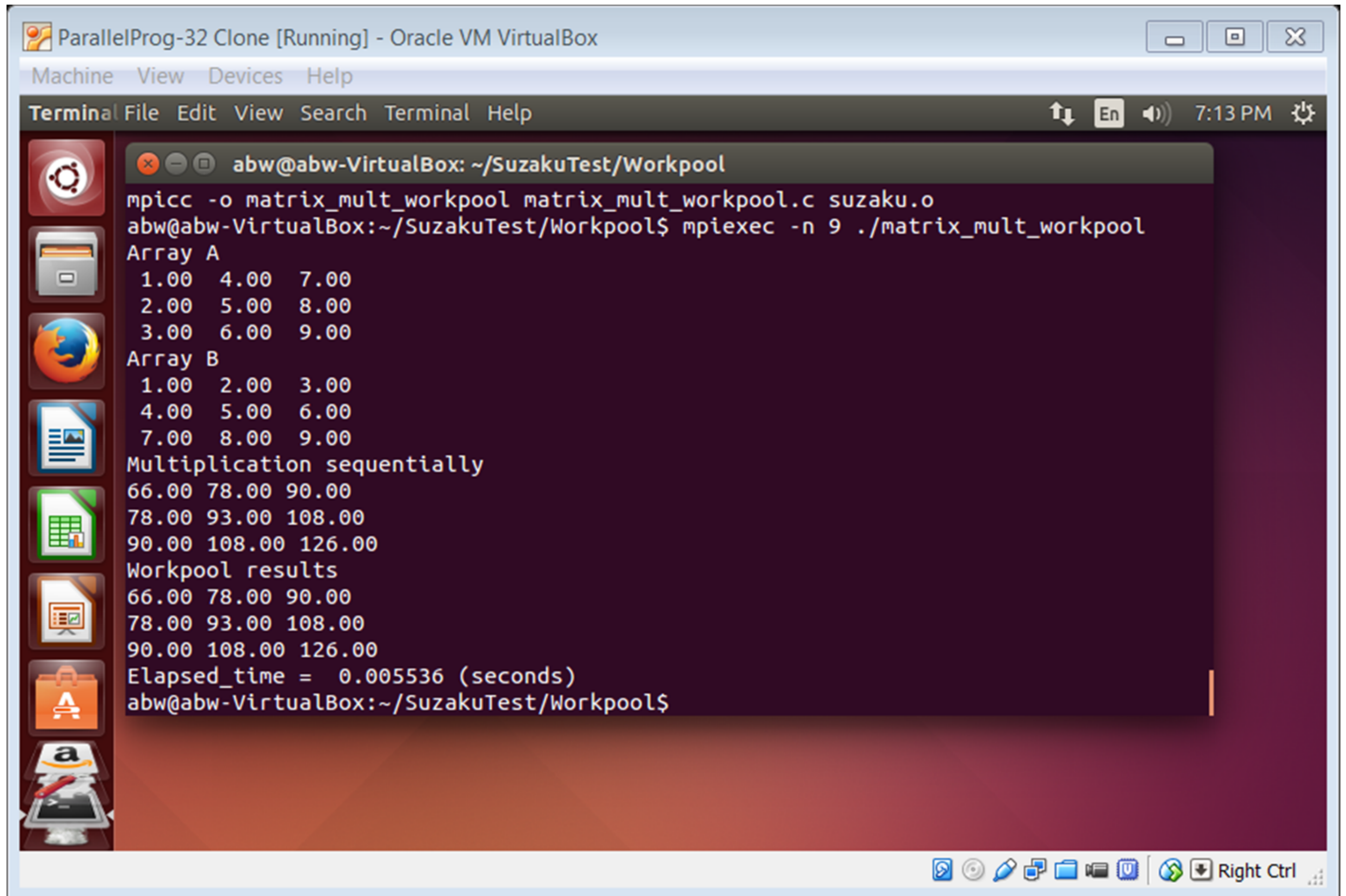
Matrix multiplication

```
// Suzaku Workpool pattern version 1 -- Application:  Matrix Multiplication. B. Wilkinson April 5, 2015

#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 6              // number of data items in each task, 3 elements of row A and 3 elements of column B
#define R 1              // number of data items in result of each task

#define N 3              // size of arrays

double A[N][N], B[N][N], C[N][N], Cseq[N][N];

// required workpool functions

void init(int *tasks, int *data_items, int *result_items) {
        *tasks = T;
        *data_items = D;
        *result_items = R;
        return;
}

void diffuse(int taskID,double output[D]) {          // uses same approach as Seeds sample but inefficient copying
arrays
                                                     // taskID used in computation
        int i;
        int a, b;
        a = taskID / N;
        b = taskID % N;
        for (i = 0; i < N; i++) {                    //Copy one row of A and one column of B into output
                output[i] = A[a][i];
                output[i+N] = B[i][b];
        }
        return;
}

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

        int i;
        output[0] = 0;
        for (i = 0; i < N; i++) {
                output[0] += input[i] * input[i+N];
        }
        return;
}

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

        int a,b;
        a = taskID / 3;
        b = taskID % 3;
        C[a][b]= input[0];
        return;
}

// additional routines used in this application

void initialize() { // initialize arrays
```

```
	int i,j;
	for (i = 0; i < N; i++){
		for(j = 0; j < N; j++) {
			A[i][j] = i + N * j + 1;
			B[i][j] = j + N * i + 1;
		}
	}
	return;
}

void seq_matrix_mult(double A[N][N], double B[N][N], double C[N][N]) {

	int i,j,k;
	for (i = 0; i < N; i++)
		for (j = 0; j < N; j++) {
			C[i][j] = 0;
			for (k = 0; k < N; k++)
				C[i][j] += A[i][k] * B[k][j];
		}
	return;
}

void print_array(double array[N][N]) { // print out an array

	int i,j;
	for (i = 0; i < N; i++){
		printf("\n");
		for(j = 0; j < N; j++) {
			printf("%5.2f ", array[i][j]);
		}
	}
	printf("\n");
	return;
}

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

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

	initialize();			// initialize input arrays
	printf("Array A");
	print_array(A);
	printf("Array B");
	print_array(B);

	seq_matrix_mult(A,B,Cseq);
	printf("Multiplication sequentially");
	print_array(Cseq);

	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

	printf("Workpool results");
	print_array(C);			// print final result
	printf("Elapsed_time = %f (seconds)\n", time2 - time1);

	SZ_Finalize();
```

```
        return 0;
}
```

Sample output (Note number of processes does not have to be the same as number of elements



3. Workpool version 2

Test program with different data types

A sample usage is in **test_workpool2.c**, shown below that demonstrates different data types that can be used with put and get:

```
// Suzaku Workpool version 2 with put and get test program
// B. Wilkinson Nov 16, 2015

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

#define T 4        // number of tasks, max = INT_MAX - 1

// workpool functions to be provided by programmer:

void init(int *tasks) {  // sets number of tasks
    *tasks = T;
    return;
}

void diffuse(int taskID) {
    int j;
    char w[] = "Hello World";
    static int x = 1234;          // only initialized first time function called
    static double y = 5678;
    double z[2][3];
    z[0][0] = 357;
    z[1][1] = 246;

    SZ_Put("w",w);
    SZ_Put("x",&x);
    SZ_Put("y",&y);
    SZ_Put("z",z);

    printf("Diffuse Task sent:     taskID = %2d, w = %s, x = %5d, y = %8.2f, z[0][0] = %8.2f, z[1][1] = %8.2f\n",taskID, w,
x, y,z[0][0],z[1][1]);

    x++;
    y++;

    return;
}

void compute(int taskID) {  // simply passing data multiplied by 10 in a different order
    char w[12] = "-----------";
    int x = 0;
    double y = 0;
    double z[2][3];
    z[0][0] = 0;
    z[1][1] = 0;

    SZ_Get("z",z);
    SZ_Get("x",&x);
    SZ_Get("w",w);
    SZ_Get("y",&y);

    printf("Compute Task received: taskID = %2d, w = %s, x = %5d, y = %8.2f, z[0][0] = %8.2f, z[1][1] = %8.2f\n",taskID,
w, x, y,z[0][0],z[1][1]);
    x = x * 10;
    y = y * 10;
    z[0][0] = z[0][0] * 10;
```

```
	z[1][1] = z[1][1] * 10;
	printf("Compute Result:	taskID = %2d, w = %s, x = %5d, y = %8.2f, z[0][0] = %8.2f, z[1][1] = %8.2f\n",taskID, w,
x, y,z[0][0],z[1][1]);

	SZ_Put("xx",&x);  // use different names for test, could have been same names
	SZ_Put("yy",&y);
	SZ_Put("zz",z);
	SZ_Put("ww",w)

	return;
}

void gather(int taskID) {   // function done by master collecting slave results. Final results computed by master
	char w[12] = "-----------";
	int x = 0;
	double y = 0;
	double z[2][3];
	z[0][0] = 0;
	z[1][1] = 0;

	SZ_Get("ww",w);
	SZ_Get("zz",z);
	SZ_Get("xx",&x);
	SZ_Get("yy",&y);

	printf("Gather Task received:  taskID = %2d, w = %s, x = %5d, y = %8.2f, z[0][0] = %8.2f, z[1][1] = %8.2f\n",taskID,
w, x, y,z[0][0],z[1][1]);

	return;
}

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)

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

	printf("number of tasks = %d\n",T);

	SZ_Parallel_begin

		SZ_Workpool2(init,diffuse,compute,gather);

	SZ_Parallel_end;		// end of parallel

	SZ_Finalize();

	return 0;
}
```

test_workpool2.c

Note how the order of put and get are not the same although they could be the same. Also the names used to identify the variables are chosen by the programmer. (They are limited to eight characters in the current implementation for simplicity.)

Sample output is given below:

ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox

```
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpiexec -n 3 test_workpool2
number of tasks = 4
Diffuse Task sent:      taskID =  0, w = Hello World, x =  1234, y =  5678.00, z[0][0] =   357.00, z[1][1] =   246.00
Diffuse Task sent:      taskID =  1, w = Hello World, x =  1235, y =  5679.00, z[0][0] =   357.00, z[1][1] =   246.00
Compute Task received:  taskID =  0, w = Hello World, x =  1234, y =  5678.00, z[0][0] =   357.00, z[1][1] =   246.00
Compute Result:         taskID =  0, w = Hello World, x = 12340, y = 56780.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Gather Task received:   taskID =  0, w = Hello World, x = 12340, y = 56780.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Compute Task received:  taskID =  1, w = Hello World, x =  1235, y =  5679.00, z[0][0] =   357.00, z[1][1] =   246.00
Compute Result:         taskID =  1, w = Hello World, x = 12350, y = 56790.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Diffuse Task sent:      taskID =  2, w = Hello World, x =  1236, y =  5680.00, z[0][0] =   357.00, z[1][1] =   246.00
Gather Task received:   taskID =  1, w = Hello World, x = 12350, y = 56790.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Compute Task received:  taskID =  2, w = Hello World, x =  1236, y =  5680.00, z[0][0] =   357.00, z[1][1] =   246.00
Compute Result:         taskID =  2, w = Hello World, x = 12360, y = 56800.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Diffuse Task sent:      taskID =  3, w = Hello World, x =  1237, y =  5681.00, z[0][0] =   357.00, z[1][1] =   246.00
Compute Task received:  taskID =  3, w = Hello World, x =  1237, y =  5681.00, z[0][0] =   357.00, z[1][1] =   246.00
Gather Task received:   taskID =  2, w = Hello World, x = 12360, y = 56800.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Compute Result:         taskID =  3, w = Hello World, x = 12370, y = 56810.00, z[0][0] =  3570.00, z[1][1] =  2460.00
Gather Task received:   taskID =  3, w = Hello World, x = 12370, y = 56810.00, z[0][0] =  3570.00, z[1][1] =  2460.00
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

Matrix Multiplication

A program implementing matrix multiplication in the same way as the Seeds sample matrix multiplication program called **matrixmult_workpool2.c** is given below:

```
// Suzaku Workpool pattern version 2 -- Application:  Matrix Multiplication
// B. Wilkinson Nov 16, 2015

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

#define T 9                // required Suzaku constant, number of tasks, max = INT_MAX - 1
#define N 3                // size of matrices

double A[N][N], B[N][N], C[N][N], Cseq[N][N];

// required workpool functions

void init(int *tasks) {
        *tasks = T;
        return;
}

void diffuse(int taskID) {       // same approach as Seeds sample but inefficient copying arrays
                                 // taskID used in computation
        int i;
        int a, b;
        double rowA[3],colB[3];

        a = taskID / N;
        b = taskID % N;
        for (i = 0; i < N; i++) {    //Copy one column of B into output, do not need to copy row of A
                colB[i] = B[i][b];
        }

        SZ_Put("rowA",A[a]);
        SZ_Put("colB",colB);

        return;
}

void compute(int taskID) {

        int i;
        double out;
        double rowA[3],colB[3];

        SZ_Get("rowA",rowA);
        SZ_Get("colB",colB);

        out = 0;
        for (i = 0; i < N; i++) {
                out += rowA[i] * colB[i];
        }

        SZ_Put("out",&out);

        return;
```

```
}

void gather(int taskID) {

        int a,b;
        double answer;
        SZ_Get("out",&answer);
        a = taskID / 3;
        b = taskID % 3;
        C[a][b]= answer;

        return;
}

// additional routines used in this application

void initialize() { // initialize arrays

        int i,j;
        for (i = 0; i < N; i++){
                for(j = 0; j < N; j++) {
                        A[i][j] = i + N * j + 1;
                        B[i][j] = j + N * i + 1;
                }
        }
        return;
}

void seq_matrix_mult(double A[N][N], double B[N][N], double C[N][N]) {

        int i,j,k;
        for (i = 0; i < N; i++)
                for (j = 0; j < N; j++) {
                        C[i][j] = 0;
                        for (k = 0; k < N; k++)
                                C[i][j] += A[i][k] * B[k][j];
                }
        return;
}

void print_array(double array[N][N]) { // print out an array

        int i,j;
        for (i = 0; i < N; i++){
                printf("\n");
                for(j = 0; j < N; j++) {
                        printf("%5.2f ", array[i][j]);
                }
        }
        printf("\n");
        return;
}

int main(int argc, char *argv[]) {
                                        // All variables declared here are in every process
        int i;
        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

        initialize();           // initialize input arrays
        printf("Array A");
        print_array(A);
        printf("Array B");
        print_array(B);

        seq_matrix_mult(A,B,Cseq);
        printf("Multiplication sequentially");
        print_array(Cseq);

        time1 = SZ_Wtime();     // record time stamp
        SZ_Parallel_begin       // start of parallel section

        SZ_Workpool2(init,diffuse,compute,gather);

        SZ_Parallel_end;        // end of parallel
        time2 = SZ_Wtime();     // record time stamp

        printf("Workpool results");
        print_array(C);         // print final result
        printf("Elapsed_time = %f (seconds)\n", time2 - time1);

        SZ_Finalize();

        return 0;
}
```

Note in Seeds it is necessary to copy each row of **A** into a separate array for the put operation but in the Suzaku version it is not necessary as elements in each row are stored in consecutive locations in the C language and we can simply refer to the pointer to the row (A[a]).

4. Workpool version 3

1. test1_workpool3.c

Program to test task queue and messaging:

```
// Suzaku Dynamic Workpool3  -- Queue test
// B. Wilkinson Nov 23, 2015

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

// workpool functions to be provided by programmer:

void init(int *T) {  // insert initial tasks in task queue

	SZ_Master {							// only master can insert tasks
		printf("Init() inserting 0 and 1 into queue task\n");
		SZ_Insert_task(0);					// add some tasks
		SZ_Insert_task(1);
	}
	return;
}

void diffuse(int task_no) {  // allows programmer to add additional information to task before sending to slave
	char message[] = "Hello world";
	char abc[] = "abc";

	if (task_no == 0) { SZ_Put("message",message); }  else {SZ_Put("message",abc);}
	printf("Diffuse, -- Next Task = %d\n",task_no);

	return;
}

void compute(int task_no) {

	int i;
	int tasks[4];
	int task;
	int slave;
	char message [20];

	for (i = 0; i < 4;i++) tasks[i] = -1;

	slave = SZ_Get_process_num();

	SZ_Get("message",message);			// get task
	printf("Slave %d -- Task received. Task = %d, message = %s\n",slave,task_no,message);

	// some computation, add new tasks

	if (task_no == 1) {			// taskID 1 generates new tasks
		tasks[0] = 6;
		tasks[1] = 7;
	}
	SZ_Put("tasks",tasks);

	return;
}

void gather(int task_no) {

	int i;
	int tasks[4];

	SZ_Get("tasks",tasks);

	printf("Gather -- Task = %d received. ",task_no);
```

```
	for (i = 0; i < 4;i++) {
		if (tasks[i] != -1) {
			SZ_Insert_task(tasks[i]);
			printf("New task %d added to queue. ",tasks[i]);
		}
	}
	printf("\n");

	return;
}

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)

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

		SZ_Workpool3(init,diffuse,compute,gather);

	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/Suzaku11_24_15

```
abw@abw-VirtualBox:~/ParallelProg/Suzaku11_24_15$ mpiexec -n 4 test1_workpool3
Workpool version 3 Nov 24, 2015
Init() inserting 0 and 1 into queue task
Diffuse, -- Next Task = 0
Diffuse, -- Next Task = 1
Slave 1 -- Task received. Task = 0, message = Hello world
Slave 2 -- Task received. Task = 1, message = abc
Gather -- Task = 1 received. New task 6 added to queue. New task 7 added to queue.
Diffuse, -- Next Task = 6
Diffuse, -- Next Task = 7
Slave 3 -- Task received. Task = 6, message = abc
Gather -- Task = 0 received.
Slave 2 -- Task received. Task = 7, message = abc
Gather -- Task = 7 received.
Gather -- Task = 6 received.
abw@abw-VirtualBox:~/ParallelProg/Suzaku11_24_15$
```

Right Ctrl

2. Shortest path problem

From page 214, *Parallel Programming Techniques and Applications*, 2nd ed. by B. Wilkinson, Prentice Hall 2005.

Sequential version, **shortest_path.c**:

```
// shortest path problem, sequential version  B. Wilkinson Nov 25, 2015
#include <stdio.h>
#include <string.h>
#define N 6                 // number of nodes
#define QSIZE 10            // size of queue

int w[N][N];                // Adjacency matrix for w
int dist[N];                //Current minimum distances
int newdist_j;

int queue[QSIZE];           // task queue
int queue_front;            // task queue index for next task to add
int queue_rear;             // task queue index for next item to remove
int q_no_tasks;             // number of items in queue

void print_dist() {
	int i;
	printf("Current minimum distances = ");
	for (i = 0; i < N; i++)
	  printf("%3d ", dist[i]);
	printf("\n");
	return;
}

int queue_insert(int taskID) {  // insert task into task queue
	int status;
	status = 0;
	if (q_no_tasks == QSIZE) {
		status = -1;            // Queue full, no task added
	} else {
		queue[queue_front] = taskID;      // Task added
		q_no_tasks = q_no_tasks + 1;
	queue_front = (queue_front + 1) % QSIZE; // front points to next free space to insert
		status = q_no_tasks;          // returns number of tasks in queue
	}
	return status;
}

int queue_remove(int *taskID) {  // remove task from task queue
	int status;
	status = 0;
	if (q_no_tasks == 0) {
	status = -1;                  // Queue empty
	} else {
		*taskID = queue[queue_rear];          // Task removed
		q_no_tasks = q_no_tasks - 1;
		queue_rear = (queue_rear + 1) % QSIZE;        // rear points to next item to remove
		status = q_no_tasks;              // returns number of tasks in queue
	}
	return status;
}

void queue_print() {  // for testing
	int i;
	printf("Contents of queue: ");
	if (q_no_tasks == 0) printf("Queue empty\n");

	for(i = 0; i < q_no_tasks; i++) {
	printf("%d ",queue[(queue_rear + i) % QSIZE] ); // print queue[(rear + i) % QSIZE]
	}
	printf("\n");
	return;
}

void queue__init() {  // initialize to zero
```

```
        int i;
        queue_front = 0;            // task queue index for next task to add
        queue_rear = 0;                     // task queue index for next item to remove
        q_no_tasks = 0;             // number of items in queue
        return;
}

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

        for (i = 0; i < N; i++) dist[i] = 99999; // initialize to greater than the max possible distance
        dist[0] = 0;                            // distance from first node to itself = zero

        for (i = 0; i < N; i++)
        for (j = 0; j < N; j++)
                w[i][j] = -1;       // initialize to no connection
        w[0][1] = 10;               // set specific connections
        w[1][2] = 8;
        w[1][3] = 13;
        w[1][4] = 24;
        w[1][5] = 51;
        w[2][3] = 14;
        w[3][4] = 9;
        w[4][5] = 17;

        printf("Adjacency matrix for w\n");
        for (i = 0; i < N; i++) {
          for (j = 0; j < N; j++)
                printf("%3d ", w[i][j]);
          printf("\n");
        }

        queue__init();
        queue_insert(0);    // insert first node 0 into queue
        queue_print();
        while (queue_remove(&i) != -1) {                // vertex i from queue
          printf("Vertex %d removed  ",i);
          queue_print();print_dist();
          for (j = 0; j < N; j++) {  // check each dest. j from vertex i, seq. order (j = 0; j < N; j++), book order  j = N-1; j >= 0; j--
                if (w[i][j] != -1) {                    // if destination j connected directly
                  newdist_j = dist[i] + w[i][j];        // distance to j thro i using current shortest distance to i
                  if (newdist_j < dist[j]) {            // update shortest distance to j if shorter
                        dist[j] = newdist_j;
                        if (j < N-1) { // do not add last vertex (destination)
                          queue_insert(j);
                          printf("New shorter distance to vertex %d found. Vertex added to queue.\n",j);
                          queue_print();print_dist();
                        }
                  }
                }
          }
        }

        return 0;
}
```

Sample output

ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Ubuntu Desktop

abw@abw-VirtualBox: ~/ParallelProg/SuzakuDev

```
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ cc shortestpath.c
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ ./a.out
Adjacency matrix for w
 -1  10  -1  -1  -1  -1
 -1  -1   8  13  24  51
 -1  -1  -1  14  -1  -1
 -1  -1  -1  -1   9  -1
 -1  -1  -1  -1  -1  17
 -1  -1  -1  -1  -1  -1
Contents of queue: 0
Vertex 0 removed  Contents of queue: Queue empty

Current minimum distances =   0 99999 99999 99999 99999 99999
New shorter distance to vertex 1 found. Vertex added to queue.
Contents of queue: 1
Current minimum distances =   0  10 99999 99999 99999 99999
Vertex 1 removed  Contents of queue: Queue empty

Current minimum distances =   0  10 99999 99999 99999 99999
New shorter distance to vertex 2 found. Vertex added to queue.
Contents of queue: 2
Current minimum distances =   0  10  18 99999 99999 99999
New shorter distance to vertex 3 found. Vertex added to queue.
Contents of queue: 2 3
Current minimum distances =   0  10  18  23 99999 99999
New shorter distance to vertex 4 found. Vertex added to queue.
Contents of queue: 2 3 4
Current minimum distances =   0  10  18  23  34 99999
Vertex 2 removed  Contents of queue: 3 4
Current minimum distances =   0  10  18  23  34  61
Vertex 3 removed  Contents of queue: 4
Current minimum distances =   0  10  18  23  34  61
New shorter distance to vertex 4 found. Vertex added to queue.
Contents of queue: 4 4
Current minimum distances =   0  10  18  23  32  61
Vertex 4 removed  Contents of queue: 4
Current minimum distances =   0  10  18  23  32  61
Vertex 4 removed  Contents of queue: Queue empty

Current minimum distances =   0  10  18  23  32  49
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$
```

Right Ctrl

Workpool version: **shortpath_workpool3.c**

```
// Suzaku Workpool version 3 -- Shortest path  B. Wilkinson Nov 25, 2015

#include <stdio.h>
#include <string.h>
#include "suzaku.h"
                        // shortest path data
#define N 6             // number of nodes
int w[N][N];            // Adjacency matrix for w. Each process will have a copy of this without needing to broadcast it
int dist[N];            // Current minimum distances. Each prcess will have their own copy
int newdist_j;

// workpool functions to be provided by programmer:

void init(int *T) {  // initialize w and dist (all processes) and insert initial tasks in task queue (master)

      int i,j;

      for (i = 0; i < N; i++) dist[i] = 99999;      // initialize to greater than the max possible distance
      dist[0] = 0;                                   // distance from first node to itself = zero

      for (i = 0; i < N; i++)
      for (j = 0; j < N; j++)
            w[i][j] = -1;                            // initialize to no connection
      w[0][1] = 10;                                  // set specific connections, matches values in book
      w[1][2] = 8;
      w[1][3] = 13;
      w[1][4] = 24;
      w[1][5] = 51;
      w[2][3] = 14;
      w[3][4] = 9;
      w[4][5] = 17;

      SZ_Master {
            SZ_Insert_task(0);                       // insert first node 0 into queue, strictly only the master needs to do this
            printf("Init() inserting 0 into task queue\n");       // only the queue in the master if used
      }
      return;
}

void diffuse(int taskID) {  // diffuse attaches the current distances

      SZ_Put("dist",dist);       // from global array dist[] in master

      printf("Diffuse Task %d sent with dist %3d %3d %3d %3d %3d %3d\n",taskID,dist[0],dist[1],dist[2],dist[3],dist[4],dist[5]);

      return;
}

void compute(int taskID) {

      int i,j;
      int new_tasks[N];  // max of N new tasks
      int slave;

      SZ_Get("dist",dist);  // update array dist[] in slave

      slave = SZ_Get_process_num();

      for (i = 0; i < N; i++) new_tasks[i] = 0;

      printf("Slave %d Task %d recvd with dist%3d %3d %3d %3d %3d
%3d\n",slave,taskID,dist[0],dist[1],dist[2],dist[3],dist[4],dist[5]);

      i = 0;
      for (j = 0; j < N; j++) {                      // check each destination j from vertex taskno, sequential order
            if (w[taskID][j] != -1) {                // if destination j connected directly
              newdist_j = dist[taskID] + w[taskID][j];          // distance to j thro i using current shortest distance to i
              if (newdist_j < dist[j]) {                        // update shortest distance to j if shorter
                  dist[j] = newdist_j;
                  if (j < N-1) { // do not add last vertex (destination)
                    new_tasks[i] = j;
```

```
                    i++;
                    printf("Slave %d Task %d New shorter dist. to vertex %d found. Vertex added to result\n",slave,taskID,j);
                }
            }
        }
    }

    printf("Slave %d Task %d Tasks generated %2d,%2d,%2d,%2d,%2d,%2d, current dist. %3d %3d %3d %3d %3d
%3d\n",slave,taskID,new_tasks[0],new_tasks[1],new_tasks[2],new_tasks[3],new_tasks[4],new_tasks[5],dist[0],dist[1],dist[2],dist[3],d
ist[4],dist[5]);

    SZ_Put("result",new_tasks);
    SZ_Put("dist",dist);

    return;
}

void gather(int taskID) {

    int i;
    int dist_recv[N];
    int new_tasks[N];  // max of N new task

    SZ_Get("result",new_tasks);       // this will only get the first added task
    SZ_Get("dist",dist_recv);

printf("Gather  Task %d Tasks received  %2d,%2d,%2d,%2d,%2d,%2d, dist. received %3d %3d %3d %3d %3d
%3d\n",taskID,new_tasks[0],new_tasks[1],new_tasks[2],new_tasks[3],new_tasks[4],new_tasks[5],dist_recv[0],dist_recv[1],dist_recv[
2],dist_recv[3],dist_recv[4],dist_recv[5]);

    for (i = 0; i < N; i++)
    if (dist_recv[i] < dist[i]) dist[i] = dist_recv[i];   // this will update dist in master. Possible received values on the smallest

    for (i = 0; i < N; i++) {
        if (new_tasks[i] != 0) {
            SZ_Insert_task(new_tasks[i]);
        }
    }


printf("Gather  Task %d current dist.  %3d %3d %3d %3d %3d %3d\n",taskID,dist[0],dist[1],dist[2],dist[3],dist[4],dist[5]);

    return;
}

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

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

        SZ_Workpool3(init,diffuse,compute,gather);

    SZ_Parallel_end;                    // end of parallel

    printf("\nFinal results: distances %3d %3d %3d %3d %3d %3d\n",dist[0],dist[1],dist[2],dist[3],dist[4],dist[5]);

    SZ_Finalize();

    return 0;
}
```

Sample output

ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal

abw@abw-VirtualBox: ~/ParallelProg/SuzakuDev

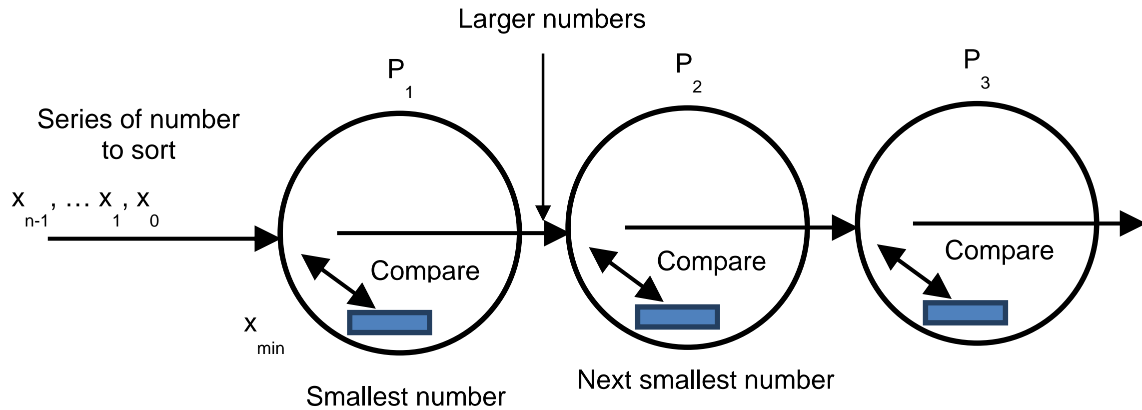
```
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ mpiexec -n 4 shortpath_workpool3
Workpool version 3 Nov 25, 2015
Init() inserting 0 into task queue
Diffuse Task 0 sent with dist   0 99999 99999 99999 99999 99999
Slave 1 Task 0 recvd with dist  0 99999 99999 99999 99999 99999
Slave 1 Task 0 New shorter dist. to vertex 1 found. Vertex added to result
Slave 1 Task 0 Tasks generated  1, 0, 0, 0, 0, 0, current dist.   0  10 99999 99999 99999 99999
Gather  Task 0 Tasks received   1, 0, 0, 0, 0, 0, dist. received   0  10 99999 99999 99999 99999
Gather  Task 0 current dist.    0  10 99999 99999 99999 99999
Diffuse Task 1 sent with dist   0  10 99999 99999 99999 99999
Slave 2 Task 1 recvd with dist  0  10 99999 99999 99999 99999
Slave 2 Task 1 New shorter dist. to vertex 2 found. Vertex added to result
Slave 2 Task 1 New shorter dist. to vertex 3 found. Vertex added to result
Slave 2 Task 1 New shorter dist. to vertex 4 found. Vertex added to result
Slave 2 Task 1 Tasks generated  2, 3, 4, 0, 0, 0, current dist.   0  10  18  23  34  61
Gather  Task 1 Tasks received   2, 3, 4, 0, 0, 0, dist. received   0  10  18  23  34  61
Gather  Task 1 current dist.    0  10  18  23  34  61
Diffuse Task 2 sent with dist   0  10  18  23  34  61
Diffuse Task 3 sent with dist   0  10  18  23  34  61
Slave 3 Task 2 recvd with dist  0  10  18  23  34  61
Slave 3 Task 2 Tasks generated  0, 0, 0, 0, 0, 0, current dist.   0  10  18  23  34  61
Diffuse Task 4 sent with dist   0  10  18  23  34  61
Slave 1 Task 3 recvd with dist  0  10  18  23  34  61
Slave 1 Task 3 New shorter dist. to vertex 4 found. Vertex added to result
Slave 1 Task 3 Tasks generated  4, 0, 0, 0, 0, 0, current dist.   0  10  18  23  32  61
Gather  Task 2 Tasks received   0, 0, 0, 0, 0, 0, dist. received   0  10  18  23  34  61
Slave 2 Task 4 recvd with dist  0  10  18  23  34  61
Slave 2 Task 4 Tasks generated  0, 0, 0, 0, 0, 0, current dist.   0  10  18  23  34  51
Gather  Task 2 current dist.    0  10  18  23  34  61
Gather  Task 3 Tasks received   4, 0, 0, 0, 0, 0, dist. received   0  10  18  23  32  61
Gather  Task 3 current dist.    0  10  18  23  32  61
Diffuse Task 4 sent with dist   0  10  18  23  32  61
Gather  Task 4 Tasks received   0, 0, 0, 0, 0, 0, dist. received   0  10  18  23  34  51
Slave 3 Task 4 recvd with dist  0  10  18  23  32  61
Slave 3 Task 4 Tasks generated  0, 0, 0, 0, 0, 0, current dist.   0  10  18  23  32  49
Gather  Task 4 current dist.    0  10  18  23  32  51
Gather  Task 4 Tasks received   0, 0, 0, 0, 0, 0, dist. received   0  10  18  23  32  49
Gather  Task 4 current dist.    0  10  18  23  32  49

Final results: distances    0  10  18  23  32  49
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$
```

Right Ctrl

5 Pipeline pattern

pipeline_sort.c

A pipeline to implement insert sort:



The basic algorithm for process *Pi* is:

```
Receive x from P_{i-1}
if (stored_number < x) {
        send stored_number to P_i
        x = stored_number;
} else send x to P_i
```

pipeline_sort.c implements this pipeline pattern:

```
// Suzaku pipeline sorting using a pipeline  B. Wilkinson Dec 3rd, 2015
#include <stdio.h>
#include <string.h>
#include <math.h>
#include <stdlib.h>
#include "suzaku.h"        // Basic Suzaku macros

#define N 1        // Size of data being sent
#define P 4        // Number of processes and number of numbers, each process only handles one number

void init(int *T,int *D,int *R) { // initialization. R not used
        *T = 4;
        *D = 1;
        //*R = 1;            // not used
        srand(999);
        return;
}

void diffuse (int taskID,double output[N]) {
        if (taskID < P) output[0] = rand()% 100;        // P numbers, a number between 0 and 99
        else output[0] = 999;                           // otherwise terminator
        return;
}

void compute(int taskID, double input[N], double output[N]) {   //  Only input[0] used in this application          static
double largest = 0;
        if (input[0] > largest) {
                output[0] = largest;         // copy current largest into send array
                largest = input[0];          // replace largest with received number
```

```
	} else {
		output[0] = input[0];			// copy received number into send array
	}
	return;
}

void gather(int taskID,double input[N]) {
	if (input[0] == 999) SZ_terminate();
	return;
}

int main(int argc, char *argv[]) {
	int p;				// p is actual number of processes when executing program
	SZ_Init(p);			// initialize MPI message-passing environment
	if (p != P)			// number of processes hardcoded
		printf("ERROR number of processes must be %d\n",P);

	SZ_Parallel_begin				// parallel section, all processes do this

		SZ_Debug();
		SZ_Pipeline(init,diffuse,compute,gather);

	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

abw@abw-VirtualBox: ~/ParallelProg/Suzaku

```
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpiexec -n 4 pipeline_sort
Debug mode displaying messages
Master sends task 0 data =  82
Slave 1 receives task 0 data =  82 and returns   0
Slave 2 receives task 2147483647 data =   0 and returns   0
Slave 3 receives task 2147483647 data =   0 and returns   0
Master receives task 2147483647,   0
Master sends task 1 data =  27
Slave 1 receives task 1 data =  27 and returns  27
Slave 2 receives task 0 data =   0 and returns   0
Slave 3 receives task 2147483647 data =   0 and returns   0
Master receives task 2147483647,   0
Master sends task 2 data =  76
Slave 1 receives task 2 data =  76 and returns  76
Slave 2 receives task 1 data =  27 and returns   0
Slave 3 receives task 0 data =   0 and returns   0
Master receives task 2147483647,   0
Master sends task 3 data =  56
Slave 1 receives task 3 data =  56 and returns  56
Slave 2 receives task 2 data =  76 and returns  27
Slave 3 receives task 1 data =   0 and returns   0
Master receives task 0,   0
Master sends task 4 data = 999
Slave 1 receives task 4 data = 999 and returns  82
Slave 2 receives task 3 data =  56 and returns  56
Slave 3 receives task 2 data =  27 and returns   0
Master receives task 1,   0
Master sends task 5 data = 999
Slave 1 receives task 5 data = 999 and returns 999
Slave 2 receives task 4 data =  82 and returns  76
Slave 3 receives task 3 data =  56 and returns  27
Master receives task 2,   0
Master sends task 6 data = 999
Slave 1 receives task 6 data = 999 and returns 999
Slave 2 receives task 5 data = 999 and returns  82
Slave 3 receives task 4 data =  76 and returns  56
Master receives task 3,  27
Master sends task 7 data = 999
Slave 1 receives task 7 data = 999 and returns 999
Slave 2 receives task 6 data = 999 and returns 999
Slave 3 receives task 5 data =  82 and returns  76
Master receives task 4,  56
Master sends task 8 data = 999
Slave 1 receives task 8 data = 999 and returns 999
Slave 2 receives task 7 data = 999 and returns 999
Slave 3 receives task 6 data = 999 and returns  82
Master receives task 5,  76
Master sends task 9 data = 999
Slave 1 receives task 9 data = 999 and returns 999
Slave 2 receives task 8 data = 999 and returns 999
Slave 3 receives task 7 data = 999 and returns 999
Master receives task 6,  82
Master sends task 10 data = 999
Slave 1 receives task 10 data = 999 and returns 999
Slave 2 receives task 9 data = 999 and returns 999
Slave 3 receives task 8 data = 999 and returns 999
Master receives task 7, 999
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

Right Ctrl

5. Generalized pattern

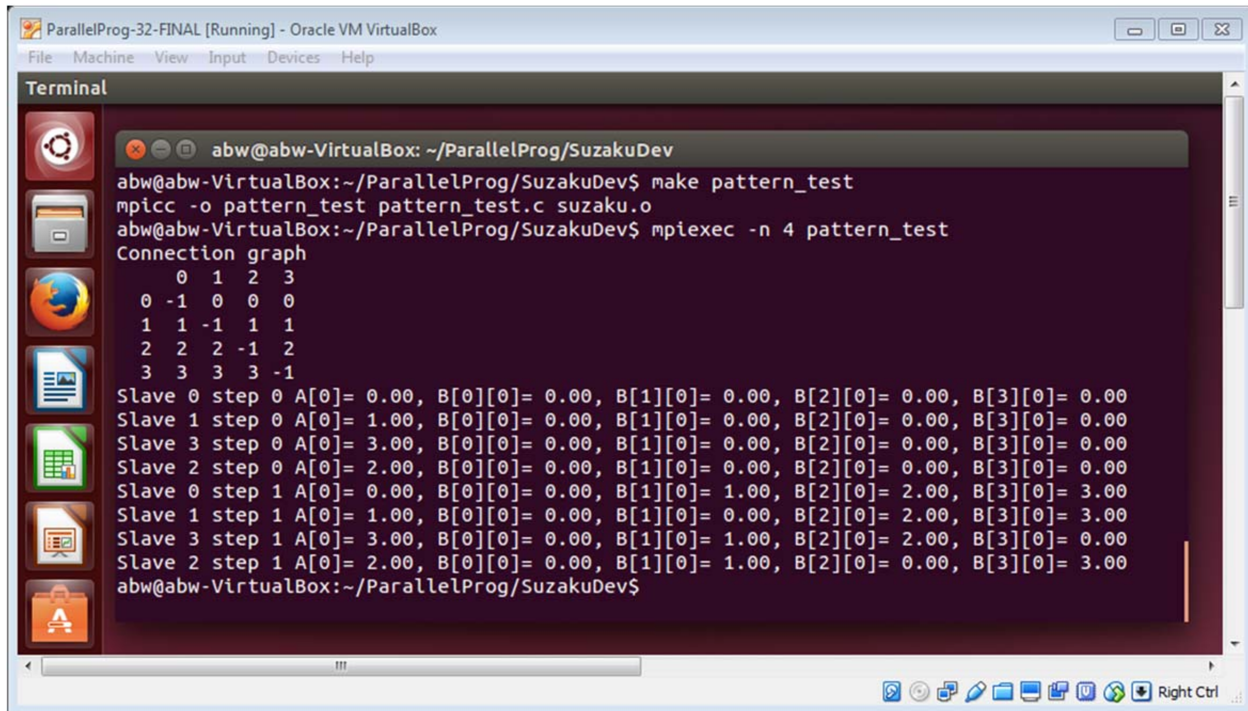
Sample programs

1. all-to-all pattern, pattern_test.c

This program simply tests the all-to-all pattern.

```
// testing generalized patterns, pattern_test.c B. Wilkinson Dec 19, 2015   Notes:  master acts as one slave
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h"        // Basic Suzaku macros

                //Declared as constants to allow static arrays for input and output
#define D 2     // # of data items in slave data.
#define P 4     // Number of processes -- this code must be run only with this number of processes

void compute(int taskID, double B[P][D], double A[D]) {   //  each slave

        printf("Slave %d step %d A[0]=%5.2f, B[0][0]=%5.2f, B[1][0]=%5.2f, B[2][0]=%5.2f, B[3][0]=%5.2f\n",
SZ_Get_process_num(), taskID,A[0],B[0][0],B[1][0],B[2][0],B[3][0]);

        return;
}

int main(int argc, char *argv[]) {
        int i,j,p;                      // p is actual number of processes when executing program
        double A[D],B[P][D];                    // A is the slave data, B holds data sent from other slaves
        int steps = 2;                          // number of time steps

        SZ_Init(p);                             // initialize MPI message-passing environment

        if (p != P) printf("ERROR Program must be run with %d processes\n",P);

        SZ_Parallel_begin                       // parallel section, all processes do this

          for (i = 0; i < D; i++) {             // all processes
                A[i] = SZ_Get_process_num();
                for (j = 0; j < P; j++){        // initialize data
                   B[j][i] = 0;
                }
          }

          SZ_Pattern_init("all-to-all",D);      // set up slave interconnections
          SZ_Print_connection_graph();          // for checking

          //SZ_Broadcast(A);                    // broadcast initial data to all slaves
                                        // not actually needed here as data is initialized in each process
          for (i = 0; i < steps; i++) {

                compute(i, B, A);               // slaves execute compute, master acts as one slave
                SZ_Generalized_send(A, B);      // sent compute results to connected slaves

          }
          SZ_Gather(A,A);                       // collect results from slaves, gather()

        SZ_Parallel_end;                        // end of parallel
        SZ_Finalize();
        return 0;
}
```

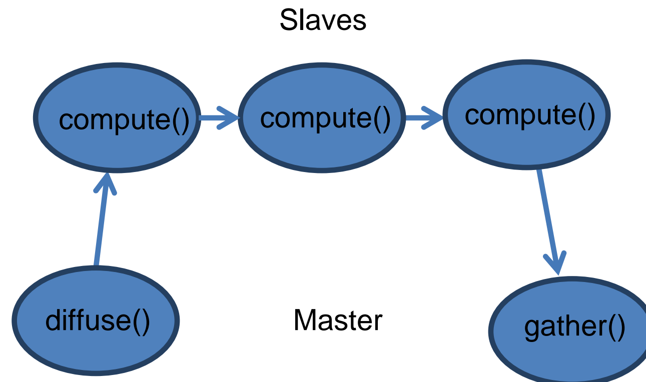
Sample output



Notice compute simply prints out the input and output arrays. The first iteration, there are at their initialized values. The second iteration shows them updated after the messaging done by **SZ_Generalized_send(A, B)**.

2. Sorting using a generalized pipeline pattern – **gen_pipeline_sort.c**

The basic pipeline is shown below described in terms of diffuse, compute and gather:



Here, the master does not act as one slave. It generates numbers and receives the final results.

```
// Sorting using a generalized pattern pipeline B. Wilkinson Dec 19, 2015.
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h"      // Basic Suzaku macros

#define D 1        // # of data items in slave data.
#define P 4        // Number of processes -- must be run only with this number of processes,
int main(int argc, char *argv[]) {
        int i, j, p,pid;                    // p is actual number of processes when executing program
        int T = 3 * P;                      // number of time steps
        double A[D];                        // data to send (D = 1).
        double B[P][D];                     // received data, from each source
        static double largest = 0;

        for (i = 0; i < D; i++) A[i] = 0;   // initialize to zero
        for (i = 0; i < P; i++)             // initialize receive so can see what received
                for (j = 0; j < D; j++)
                        B[i][j] = -999;

        srand(1);                                   // initialize rand()

        SZ_Init(p);                                 // initialize MPI message-passing environment
        if (p != P)                                 // number of processes hardcoded
                printf("ERROR number of processes must be %d\n",P);

        SZ_Parallel_begin                           // parallel section, all processes do this

          SZ_Pattern_init("pipeline",D);            // set up slave interconnections
          SZ_Print_connection_graph();              // for checking

          for (i = 0; i < T; i++) {

                pid = SZ_Get_process_num();         // identify process

                if (pid ==0)  {             // master generates next number to sort, ends with a terminator

                        if (i < P) A[0] = rand()% 100;          // P numbers, a number between 0 and 99
                        else A[0] = 999;                        // otherwise terminator
                        printf("Master sends %3.0f and receives %3.0f\n",A[0],B[0][0]);
```

```
        } else {                        // slaves execute compute, using B to create A.

            if (B[0][0] > largest) {
                A[0] = largest;    // copy current largest into send array
                largest = B[0][0];// replace largest with received number
            } else {
                A[0] = B[0][0];          // copy received number into send array
            }

        }

        SZ_Generalized_send(A,B);       // sent results, includes master to slave, slave to master
        SZ_Barrier();                   // wait for every process to complete

    }

    SZ_Parallel_end;                    // end of parallel

    SZ_Finalize();

    return 0;
}
```

Notice A and B are static arrays to match the generalized send routine. The program could have been written with specific diffuse, compute and gather routines.

Sample output:

```
ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox
File  Machine  View  Input  Devices  Help
Terminal
abw@abw-VirtualBox: ~/ParallelProg/SuzakuDev
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ mpiexec -n 4 gen_pipeline_sort
Connection graph
      0  1  2  3
  0 -1  0 -1 -1
  1 -1 -1  0 -1
  2 -1 -1 -1  0
  3  0 -1 -1 -1
Master sends  83 and receives -999
Master sends  86 and receives -999
Master sends  77 and receives -999
Master sends  15 and receives -999
Master sends 999 and receives    0
Master sends 999 and receives    0
Master sends 999 and receives    0
Master sends 999 and receives   15
Master sends 999 and receives   77
Master sends 999 and receives   83
Master sends 999 and receives   86
Master sends 999 and receives  999
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$
```

3. Stencil pattern – **gen_heat.c**

The following solve the two-dimensional heat distribution problem. For simplicity, only 16 points are used and one of 16 processes for each point. The approach can be be extended to have each process handle multiple points. This is left as an exercise.

```
// Basic heat distribution program to demostrate synchronous stencil program. gen_heat.c    B. Wilkinson Dec 28, 2015
// simplistic version with each process doing one point

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include "suzaku.h"          // Basic Suzaku macros

#define D 1         // # of data items in slave data
#define P 16        // Number of processes -- this code must be run only with this number of processes
#define N 6         // Number of pts in each dimension, to include border 6 x 6
#define M 4         // Number of pts in each dimension, not including border 4 x 4

int main(int argc, char *argv[])  {
        int i,j,x,y,t;                          // loop counters
        int T = 100;                            // time period
        int p, pid;

        double pts[N][N];                   // array of points to include fixed borders
        double A[1];                            // point being computed in slave, output array
        double B[P][D];                         // input array
        double temp[M][M];                      // hardcoded for 4 x 4

        double pts_seq[2][N][N];            // array to do computation sequentailly.
        int current = 0;
        int next = 1;

        SZ_Init(p);                             // initialize MPI message-passing environment
        if (p != P) printf("ERROR Program must be run with %d processes\n",P);
        printf("Number of points in each dimension = %d\n",N);
        printf("Number of time steps = %d\n",T);

/* ------------------------- Set up inital values ---------------*/
        for(i = 0; i < N; i++)              // load inital values into array
                for(j = 0; j < N; j++)      // border and inner points = 20
                        pts[i][j] = 20;         // note C row major order, row i, col j
        for(i = 2; i < N-2; i++)
                pts[0][i] = 100.0;          // top row = 100

        printf("Initial numbers");          // print numbers
        for(i = 0; i < N; i++)
                for(j = 0; j < N; j++) {
                        if (j == 0) printf("\n");
                        printf("%7.2f",pts[i][j]);
                }
        printf("\n");

        // compute values sequentially to check with parallel result, done using Jacobi iteration

        for(i = 0; i < N; i++)              // load inital values into array
        for(j = 0; j < N; j++) {
                pts_seq[current][i][j] = pts[i][j];
                pts_seq[next][i][j] = pts[i][j];
        }
        for (t=0; t < T; t++) {             // do computation sequentially, using Jacobi iteration
          for (i=1; i < N-1; i++)
          for (j=1; j < N-1; j++)
                pts_seq[next][i][j] = 0.25 * (pts_seq[current][i-1][j] + pts_seq[current][i+1][j] + pts_seq[current][i][j-1] +
pts_seq[current][i][j+1]);
          current = next;
          next = 1 - current;
        }

/* ------------------------Computation----------------------------------*/

        SZ_Parallel_begin           // parallel section, all processes do this

                SZ_Pattern_init("stencil",D);// set up slave interconnections
```

```
		SZ_Broadcast(pts);			// synchronous, includes a barrier
							// Set up initial values in each slave
		pid = SZ_Get_process_num();
		x = pid / M;				// row, hardcoded for 16 processes 4 x 4
		y = pid % M;				// column
		i = x + 1;				// location in pts[][]
		j = y + 1;
		A[0] = pts[i][j];			// copy location in pts[][] into A[0]
		B[0][0] = pts[i][j-1];			// left
		B[1][0] = pts[i][j+1];			// right
		B[2][0] = pts[i-1][j];			// up
		B[3][0] = pts[i+1][j];			// down

		for (t = 0; t < T; t++) {		// compute values over time T

			A[0] = 0.25 * (B[0][0] + B[1][0] + B[2][0] + B[3][0]);	// slaves execute computation,
										\\ master acts as one slave
			SZ_Generalized_send(A,B); // sent compute results in A to B in connected slaves
		}

		SZ_Gather(A,temp);			// collect results from slaves (A), into array temp, gather()

	SZ_Parallel_end;			// end of parallel

/* ------------------------ Results ----------------------------------*/

	for (x = 0; x < N; x++)  {		// update inside points
		for(y = 0; y < N; y++) {
		   if ((x > 0) && (x < N-1) && (y > 0) && (y < N-1)) {  // inside point
			i = x - 1;
			j = y - 1;
			pts[x][y] = temp[i][j];
		   }
		}
	}
	printf("Final numbers");		// print numbers
	for (i = 0; i < N; i++) {
		for(j = 0; j < N; j++) {
			if (j == 0) printf("\n");
			printf("%7.2f",pts[i][j]);
		}
	}
	printf("\n");

	int error = 0;				// check sequential and parallel versions give same answers
	for (i = 0; i < N; i++) {
		for (j = 0; j < N; j++)   {
		  if ((pts[i][j] - pts_seq[current][i][j] > 0.001) || (pts_seq[current][i][j] - pts[i][j] > 0.001))
			{ error = -1; break;}
		}
		if (error == -1) break;
	}

	if (error == -1) printf("ERROR, sequential and parallel versions give different answers\n");
	else printf("Sequential and parallel versions give same answers within +-0.001\n");

	SZ_Finalize();

	return 0;
}
```

Sample output:

ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox

File Machine View Input Devices Help

Terminal

abw@abw-VirtualBox: ~/ParallelProg/SuzakuDev

```
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ make gen_heat
mpicc -o gen_heat gen_heat.c suzaku.o -lm
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$ mpiexec -n 16 gen_heat
Number of points in each dimension = 6
Number of time steps = 100
Initial numbers
  20.00  20.00 100.00 100.00  20.00  20.00
  20.00  20.00  20.00  20.00  20.00  20.00
  20.00  20.00  20.00  20.00  20.00  20.00
  20.00  20.00  20.00  20.00  20.00  20.00
  20.00  20.00  20.00  20.00  20.00  20.00
  20.00  20.00  20.00  20.00  20.00  20.00
Final numbers
  20.00  20.00 100.00 100.00  20.00  20.00
  20.00  31.21  56.36  56.36  31.21  20.00
  20.00  28.48  37.88  37.88  28.48  20.00
  20.00  24.85  28.79  28.79  24.85  20.00
  20.00  22.12  23.64  23.64  22.12  20.00
  20.00  20.00  20.00  20.00  20.00  20.00
Sequential and parallel versions give same answers within +-0.001
abw@abw-VirtualBox:~/ParallelProg/SuzakuDev$
```

Right Ctrl

4 Printout of patterns – gen_connect_test.c

The following simply prints out the three patterns implemented.

```
// testing generalized graph  gen_connect_test.c   B. Wilkinson Dec 28, 2015
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "suzaku.h"        // Basic Suzaku macros
#define D 2                // # of data items in slave data.

int main(int argc, char *argv[]) {
        int p;                          // p is actual number of processes when executing program

        SZ_Init(p);                            // initialize MPI message-passing environment

        SZ_Parallel_begin                      // parallel section, all processes do this
                SZ_Pattern_init("all-to-all",D);   // set up slave interconnections
                SZ_Master { printf("all-to all pattern\n"); }
                SZ_Print_connection_graph();            // for checking

                SZ_Pattern_init("pipeline",D);     // set up slave interconnections
                SZ_Master { printf("pipeline pattern\n"); }
                SZ_Print_connection_graph();            // for checking

                SZ_Pattern_init("stencil",D);      // set up slave interconnections
                SZ_Master { printf("stencil\n"); }
                SZ_Print_connection_graph();            // for checking

        SZ_Parallel_end;                       // end of parallel
        SZ_Finalize();

        return 0;
}
```

Sample output: