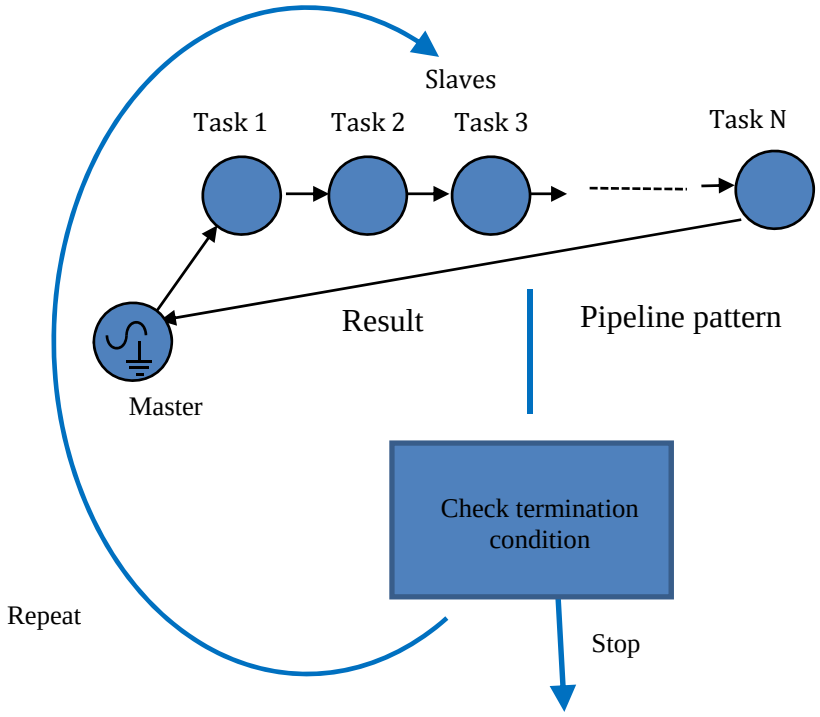
# SUZAKU Pattern Programming Framework Specification

## 5 - Pipeline Pattern Version 1

B. Wilkinson, March 17, 2016

### 5.1 Iterative synchronous pipeline pattern

The iterative synchronous pipeline pattern has been implemented as shown below:

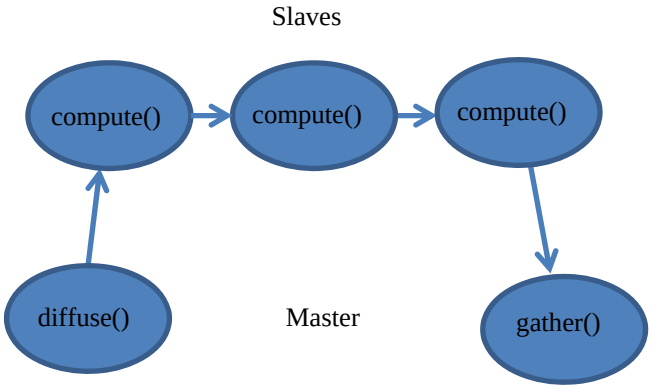



Iterative synchronous pipeline pattern

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. 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.

### 5.2 Suzaku Pipeline Routines

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:




Suzaku pipeline 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.

The signatures art the same as for 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])**

### 5.3 Pipeline Pattern Signature

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[])  )
```

### 5.4 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.

### 5.5 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 and sets a flag in SZ_Pipeline to enable debug print statements. 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.

Using above approach could be used in other patterns rather than provide two separate routines as in the workpool version 1 although this is not implemented yet and would not be as efficient.

### 5.6 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*

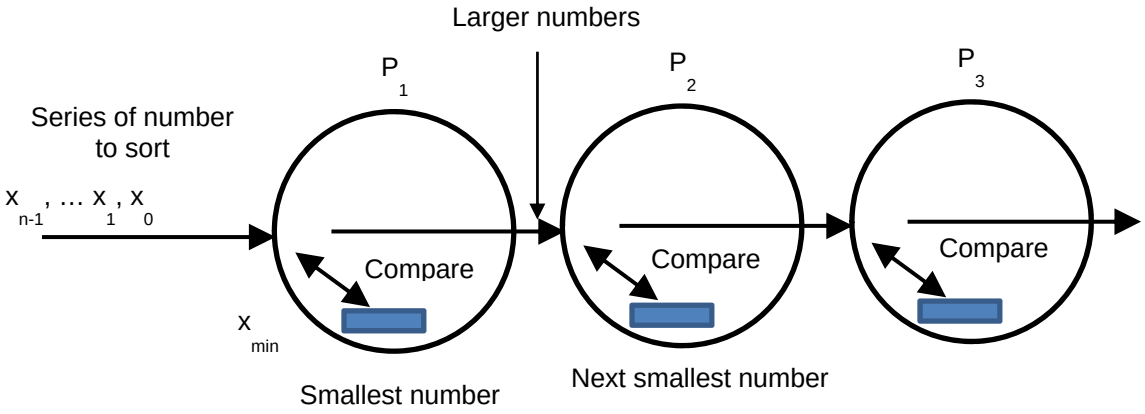
## 5.7 Compilation and Execution

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

# Sample programs

## pipeline_sort.c

A pipeline to implement insert sort:




The basic algorithm for process *Pi* is:

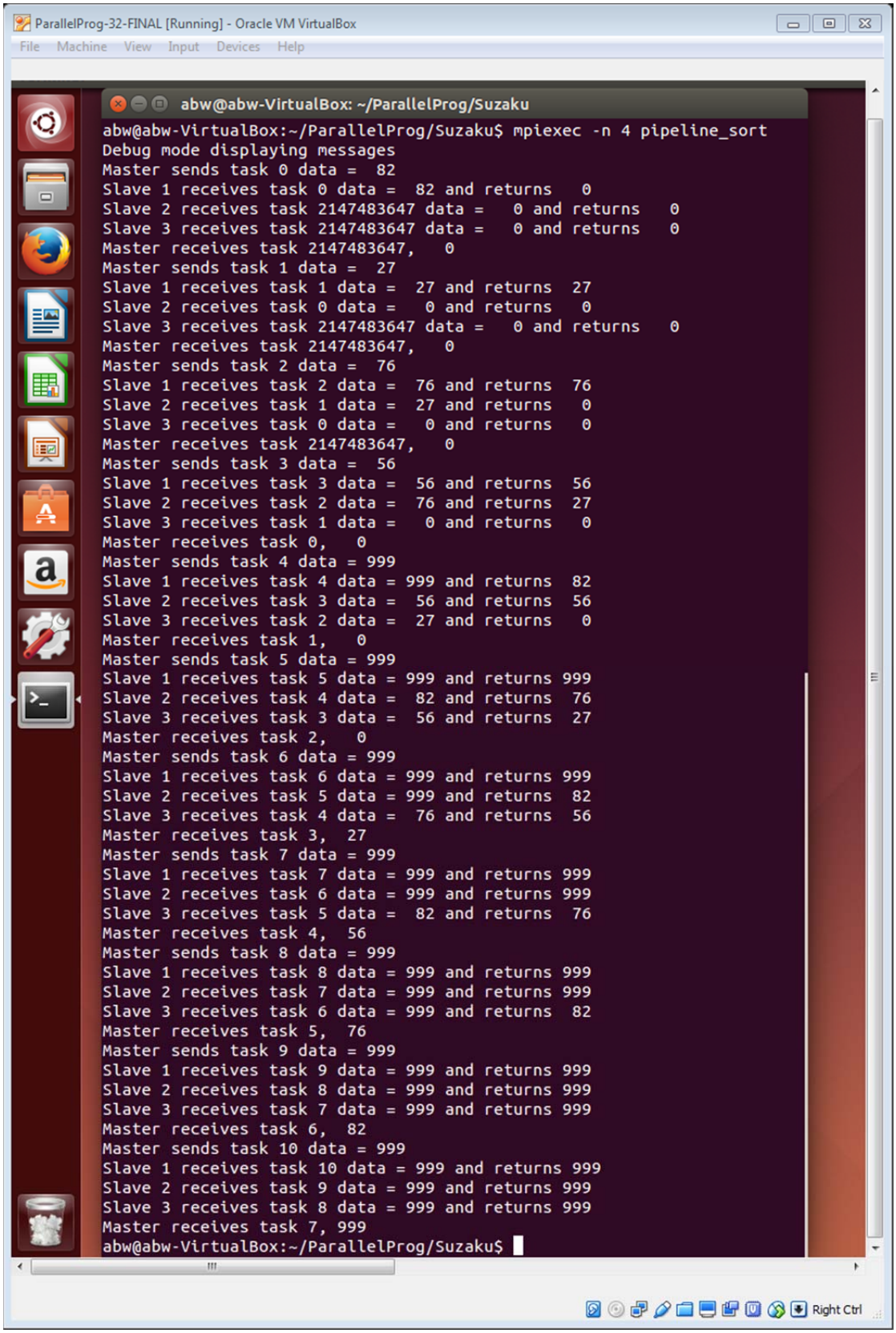
```
Receive x from Pi-1
if (stored_number < x) {
        send stored_number to Pi
        x = stored_number;
} else send x to Pi
```

**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