

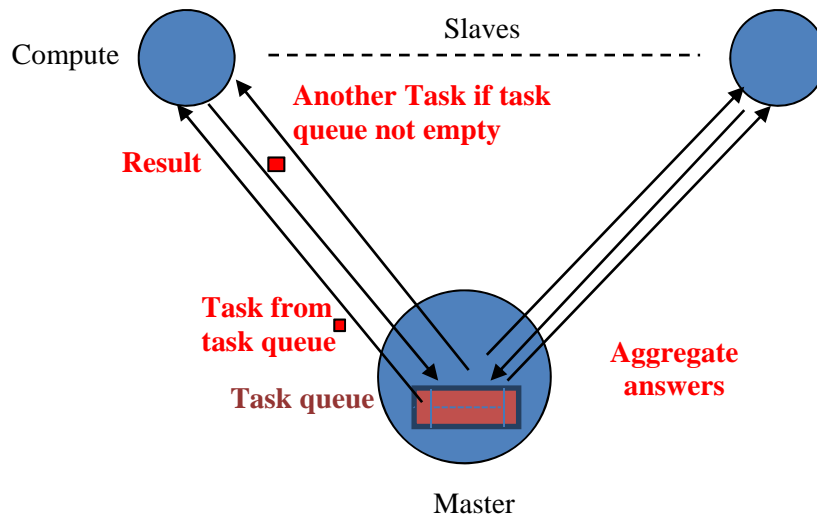
SUZAKU Pattern Programming Framework Specification

2 - Workpool Pattern Version 1

B. Wilkinson, March 17, 2016

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.

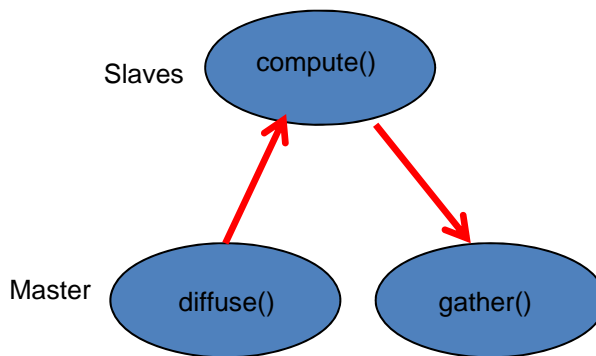
Workpool with a task queue

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 deposits 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



Message passing done by framework

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
```

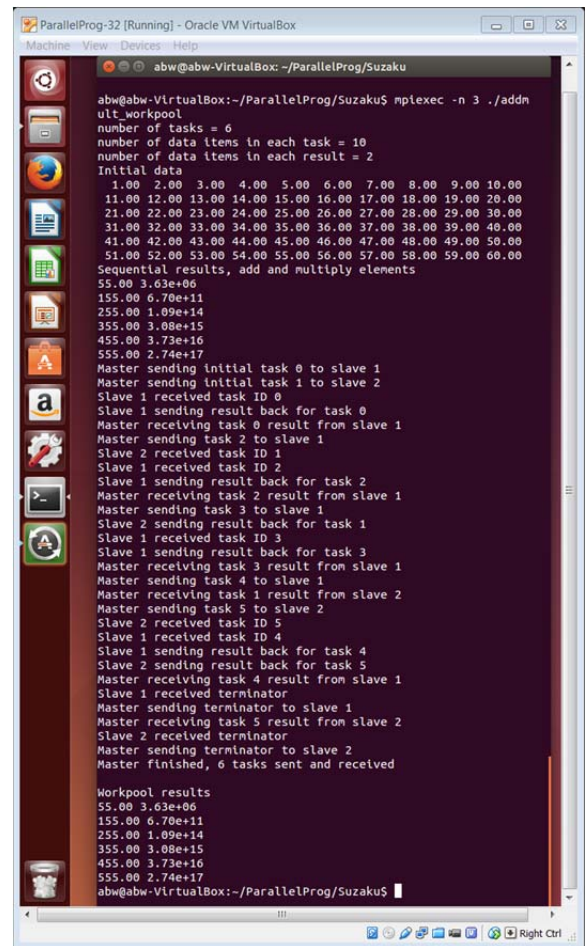
To execute **prog1**, issue the command:

```
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 slaves and results returned. This version is called **SZ_Workpool_debug()** and can be found in **suzaku.c**. To rename **SZ_Workpool()** in the application code to **SZ_Workpool_debug()** and recompile.



```
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mplexec -n 3 ./addn
ult_workpool
number of tasks = 6
number of data items in each task = 10
number of data items in each result = 2
Initial data
 1.00  2.00  3.00  4.00  5.00  6.00  7.00  8.00  9.00 10.00
11.00 12.00 13.00 14.00 15.00 16.00 17.00 18.00 19.00 20.00
21.00 22.00 23.00 24.00 25.00 26.00 27.00 28.00 29.00 30.00
31.00 32.00 33.00 34.00 35.00 36.00 37.00 38.00 39.00 40.00
41.00 42.00 43.00 44.00 45.00 46.00 47.00 48.00 49.00 50.00
51.00 52.00 53.00 54.00 55.00 56.00 57.00 58.00 59.00 60.00
Sequential results, add and multiply elements
55.00 3.63e+06
155.00 6.70e+11
255.00 1.09e+14
355.00 3.08e+15
455.00 3.73e+16
555.00 2.74e+17
Master sending initial task 0 to slave 1
Master sending initial task 1 to slave 2
Slave 1 received task ID 0
Slave 1 sending result back for task 0
Master receiving task 0 result from slave 1
Master sending task 2 to slave 1
Slave 2 received task ID 1
Slave 1 received task ID 2
Slave 1 sending result back for task 2
Master receiving task 2 result from slave 1
Master sending task 3 to slave 1
Slave 2 sending result back for task 1
Slave 1 received task ID 3
Slave 1 sending result back for task 3
Master receiving task 3 result from slave 1
Master sending task 4 to slave 1
Master receiving task 1 result from slave 2
Master sending task 5 to slave 2
Slave 2 received task ID 5
Slave 1 received task ID 4
Slave 1 sending result back for task 4
Slave 2 sending result back for task 5
Master receiving task 4 result from slave 1
Slave 1 received terminator
Master sending terminator to slave 1
Master receiving task 5 result from slave 2
Slave 2 received terminator
Master sending terminator to slave 2
Master finished, 6 tasks sent and received

Workpool results
55.00 3.63e+06
155.00 6.70e+11
255.00 1.09e+14
355.00 3.08e+15
455.00 3.73e+16
555.00 2.74e+17
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

to
use,

Sample output with debug messages

Sample programs

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

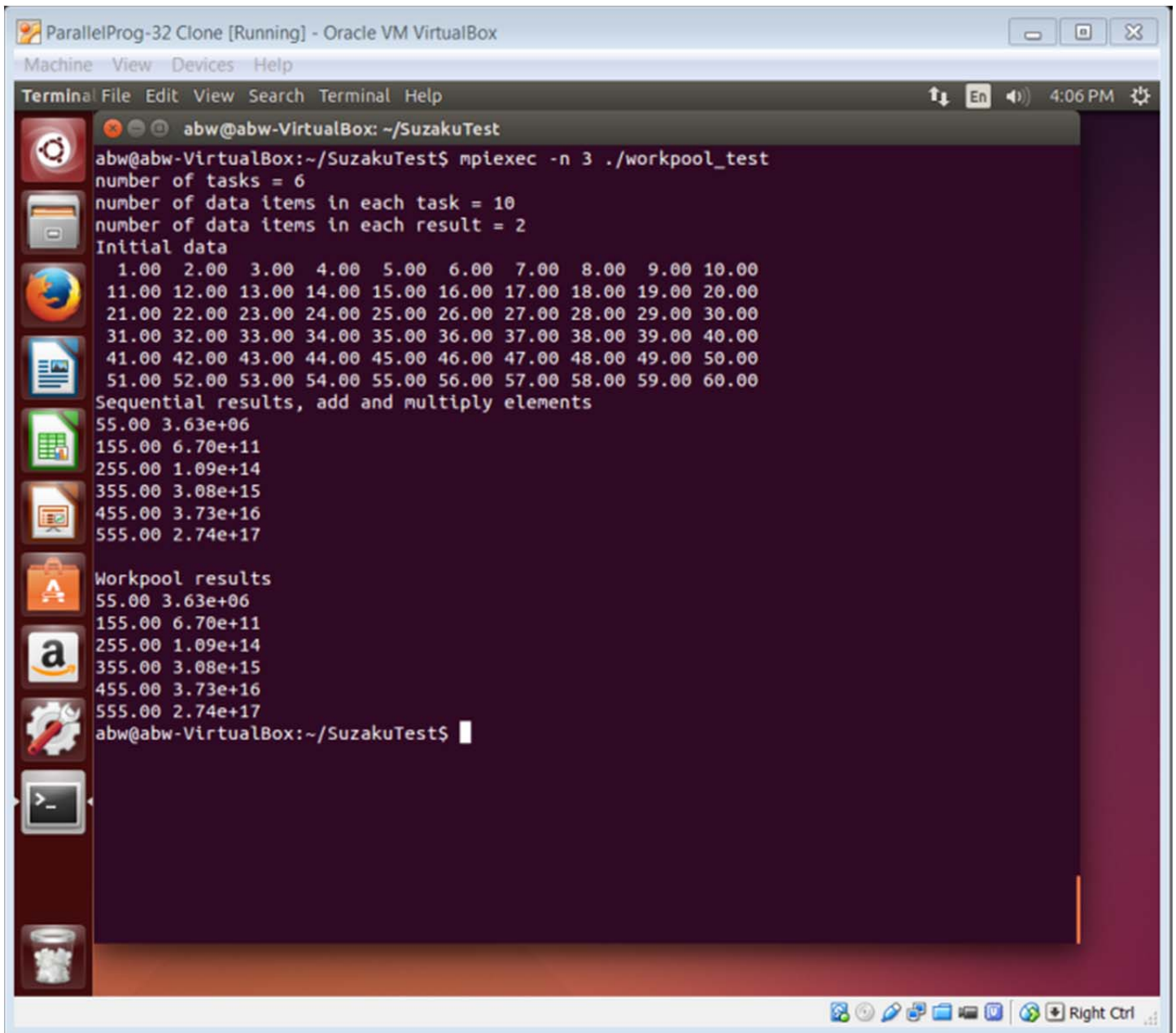
    SZ_Workpool(init,diffuse,compute,gather);

    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;
}

```

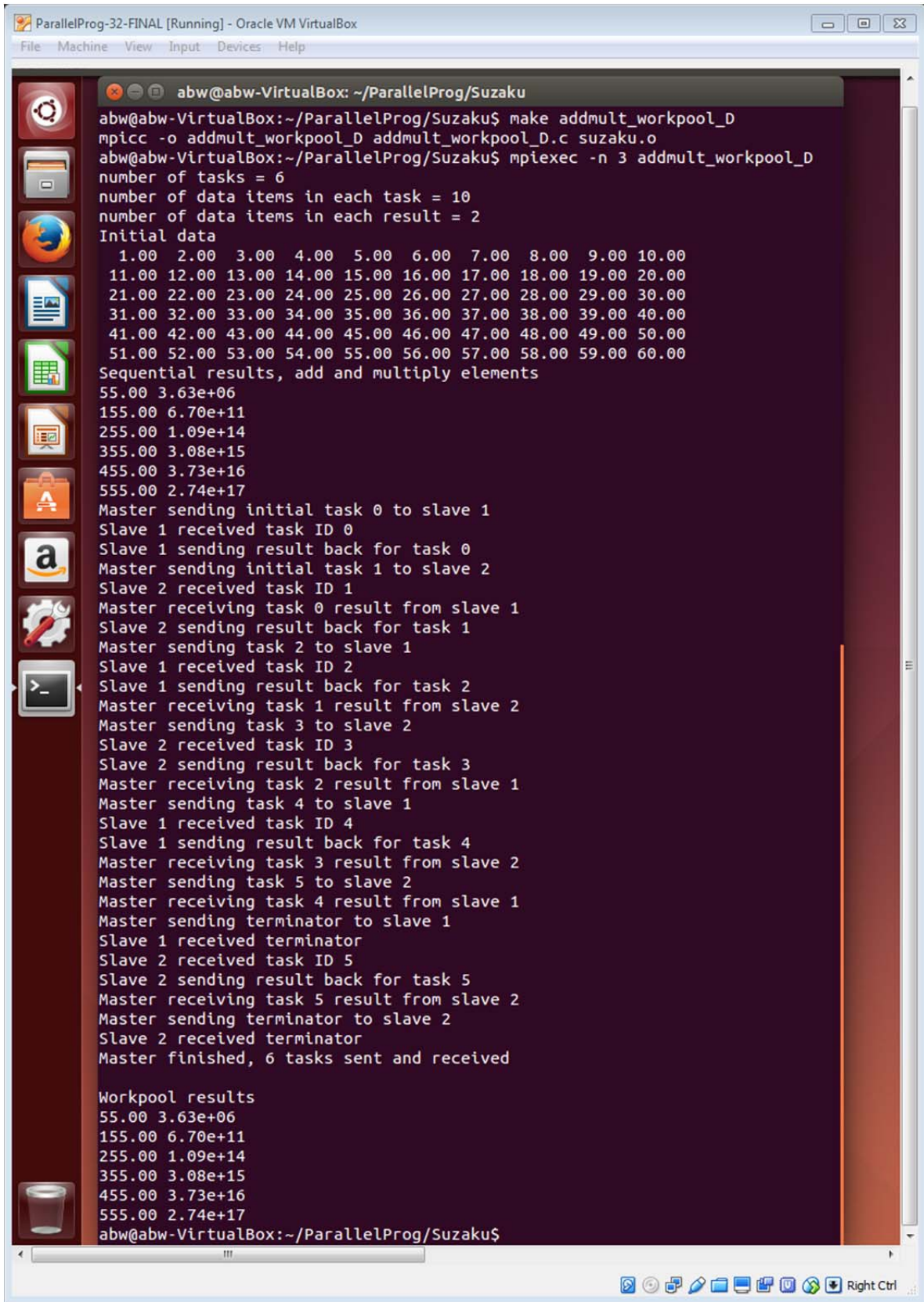

Sample output



```
ParallelProg-32 Clone [Running] - Oracle VM VirtualBox
Machine View Devices Help
Terminal File Edit View Search Terminal Help
abw@abw-VirtualBox: ~/SuzakuTest
abw@abw-VirtualBox:~/SuzakuTest$ mpiexec -n 3 ./workpool_test
number of tasks = 6
number of data items in each task = 10
number of data items in each result = 2
Initial data
 1.00  2.00  3.00  4.00  5.00  6.00  7.00  8.00  9.00 10.00
11.00 12.00 13.00 14.00 15.00 16.00 17.00 18.00 19.00 20.00
21.00 22.00 23.00 24.00 25.00 26.00 27.00 28.00 29.00 30.00
31.00 32.00 33.00 34.00 35.00 36.00 37.00 38.00 39.00 40.00
41.00 42.00 43.00 44.00 45.00 46.00 47.00 48.00 49.00 50.00
51.00 52.00 53.00 54.00 55.00 56.00 57.00 58.00 59.00 60.00
Sequential results, add and multiply elements
55.00 3.63e+06
155.00 6.70e+11
255.00 1.09e+14
355.00 3.08e+15
455.00 3.73e+16
555.00 2.74e+17
Workpool results
55.00 3.63e+06
155.00 6.70e+11
255.00 1.09e+14
355.00 3.08e+15
455.00 3.73e+16
555.00 2.74e+17
abw@abw-VirtualBox:~/SuzakuTest$
```

Version with debug messages

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$ make addmult_workpool_D
mpicc -o addmult_workpool_D addmult_workpool_D.c suzaku.o
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpiexec -n 3 addmult_workpool_D
number of tasks = 6
number of data items in each task = 10
number of data items in each result = 2
Initial data
 1.00  2.00  3.00  4.00  5.00  6.00  7.00  8.00  9.00 10.00
11.00 12.00 13.00 14.00 15.00 16.00 17.00 18.00 19.00 20.00
21.00 22.00 23.00 24.00 25.00 26.00 27.00 28.00 29.00 30.00
31.00 32.00 33.00 34.00 35.00 36.00 37.00 38.00 39.00 40.00
41.00 42.00 43.00 44.00 45.00 46.00 47.00 48.00 49.00 50.00
51.00 52.00 53.00 54.00 55.00 56.00 57.00 58.00 59.00 60.00
Sequential results, add and multiply elements
55.00 3.63e+06
155.00 6.70e+11
255.00 1.09e+14
355.00 3.08e+15
455.00 3.73e+16
555.00 2.74e+17
Master sending initial task 0 to slave 1
Slave 1 received task ID 0
Slave 1 sending result back for task 0
Master sending initial task 1 to slave 2
Slave 2 received task ID 1
Master receiving task 0 result from slave 1
Slave 2 sending result back for task 1
Master sending task 2 to slave 1
Slave 1 received task ID 2
Slave 1 sending result back for task 2
Master receiving task 1 result from slave 2
Master sending task 3 to slave 2
Slave 2 received task ID 3
Slave 2 sending result back for task 3
Master receiving task 2 result from slave 1
Master sending task 4 to slave 1
Slave 1 received task ID 4
Slave 1 sending result back for task 4
Master receiving task 3 result from slave 2
Master sending task 5 to slave 2
Master receiving task 4 result from slave 1
Master sending terminator to slave 1
Slave 1 received terminator
Slave 2 received task ID 5
Slave 2 sending result back for task 5
Master receiving task 5 result from slave 2
Master sending terminator to slave 2
Slave 2 received terminator
Master finished, 6 tasks sent and received

Workpool results
55.00 3.63e+06
155.00 6.70e+11
255.00 1.09e+14
355.00 3.08e+15
455.00 3.73e+16
555.00 2.74e+17
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

This version could be used for educational purposes.

2. Monte Carlo Pi calculation

MontePi_workpool.c:

```
// 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
```

```
// global 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)
    clock_t time1, time2; // use clock 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 = clock(); // record time stamp
    SZ_Parallel_begin // start of parallel section

    SZ_Workpool(init,diffuse,compute,gather);

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

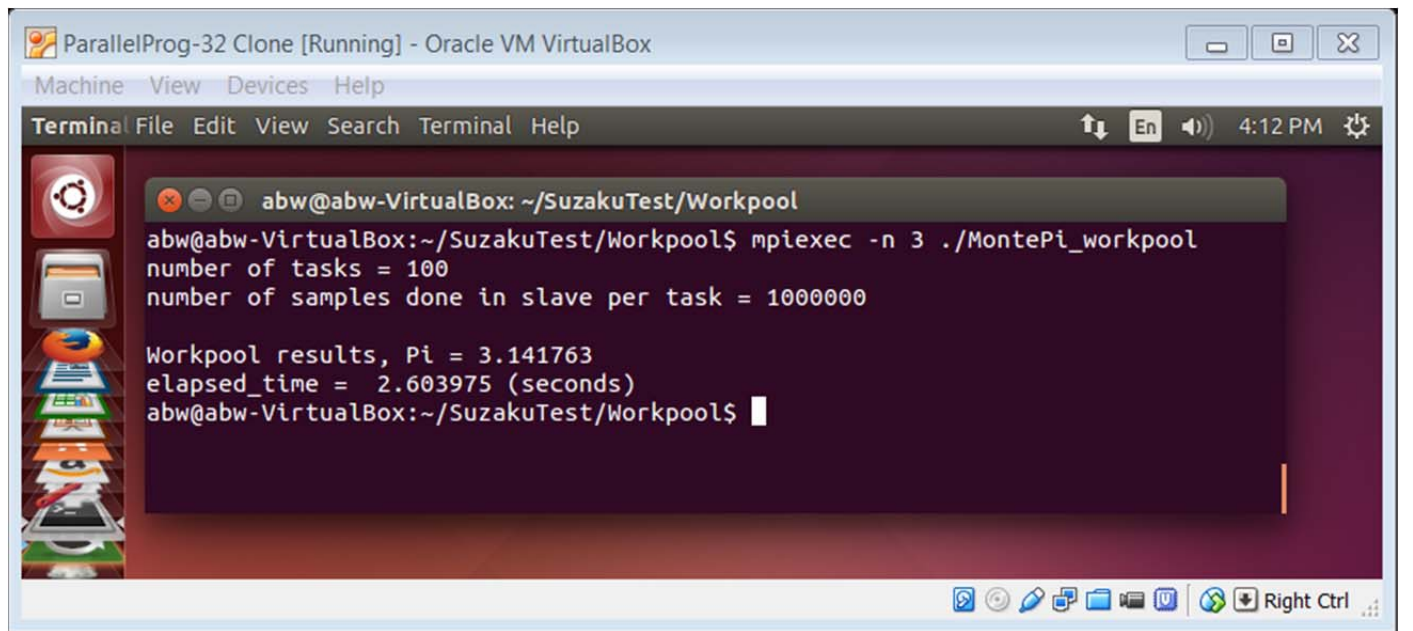
    get_pi(); // calculate final result
    printf("elapsed_time =%t%lf (seconds)\n", (double)(time2 - time1)/CLOCKS_PER_SEC);

    SZ_Finalize();

    return 0;
}

```

Sample output



```

ParallelProg-32 Clone [Running] - Oracle VM VirtualBox
Machine View Devices Help
Terminal File Edit View Search Terminal Help 4:12 PM
abw@abw-VirtualBox: ~/SuzakuTest/Workpool
abw@abw-VirtualBox:~/SuzakuTest/Workpool$ mpiexec -n 3 ./MontePi_workpool
number of tasks = 100
number of samples done in slave per task = 1000000

Workpool results, Pi = 3.141763
elapsed_time = 2.603975 (seconds)
abw@abw-VirtualBox:~/SuzakuTest/Workpool$

```

3 Matrix multiplication

matrixmult_workpool.c

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

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

#include "suzaku.h"

// required Suzaku constants
#define T 9          // 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)
    clock_t 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 = clock(); // record time stamp
    SZ_Parallel_begin // start of parallel section

    SZ_Workpool(init,diffuse,compute,gather);

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

    printf("Workpool results");
    print_array(C); // print final result
    printf("Elapsed_time =\t%lf (seconds)\n", (double)(time2 - time1)/CLOCKS_PER_SEC);
}

```

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

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

