

SUZAKU Pattern Programming Framework Specification

3 - Workpool Version 2

B. Wilkinson April 5, 2016

A version of the Suzaku workpool has been implemented that mirrors the interface in Seeds by using “put” routines to pack data into tasks and results and “get” routines to retrieve the data. Now the data can be constructed to be of multiple items of different types and sizes. 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 than version 1 but is more powerful and more elegant to use. It is basis of the dynamic workpool described in Section 4.

3.1 Workpool routines

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: The data pointed to by *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 into 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: The data pointed to by ***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 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 taskID)

Parameter:

int taskID Input parameter for the task ID for the associated task. , provided by the framework (from zero onwards incremented each diffuse is called).

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 compute(int taskID)
```

Parameter:

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

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 taskID)
```

Parameter:

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

3.2 Signature of 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
*gather	Function pointer to gather function

3.3 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 always 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”).

3.4 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.

Suggestion when debugging Suzaku code. One can get strange error messages that appear to relate to **suzaku.c** when compiling faulty application code. The errors are not in **suzaku.c**. They are often caused by missing parentheses and errors in the application code that then cause the code on **suzaku.c** to compile wrongly. It is suggested in these cases to comment out the Suzaku routines in the application code to see what exactly is erroneous in the application code.

Sample programs

1. Test program with different data types

A sample program `test_workpool2.c` shown below 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] =
%8.2fn",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] =
%8.2fn",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();

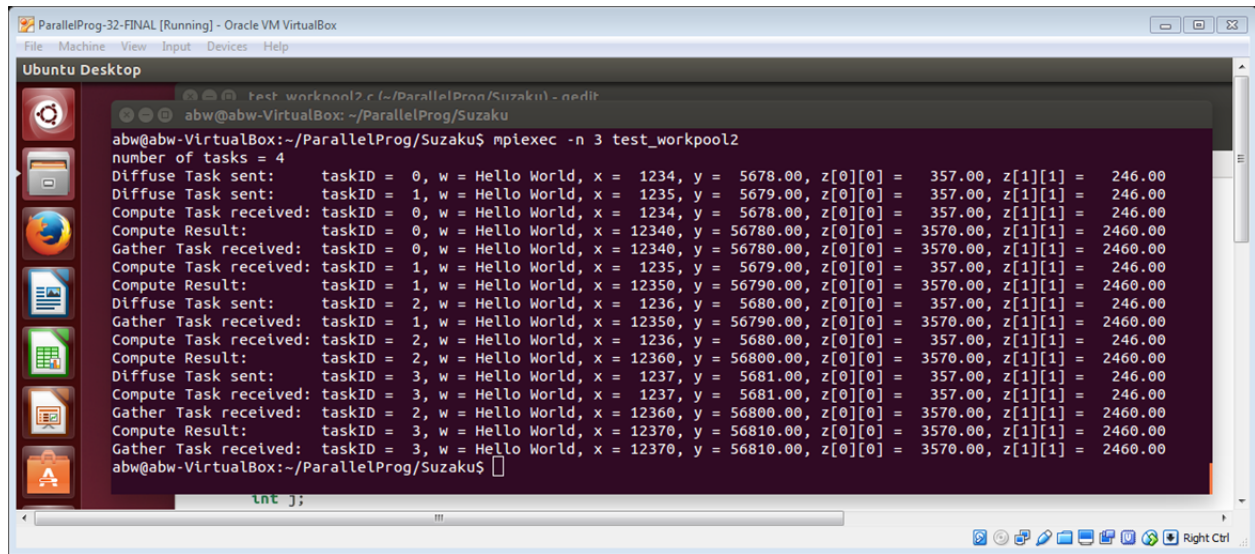
return 0;
}

```

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:

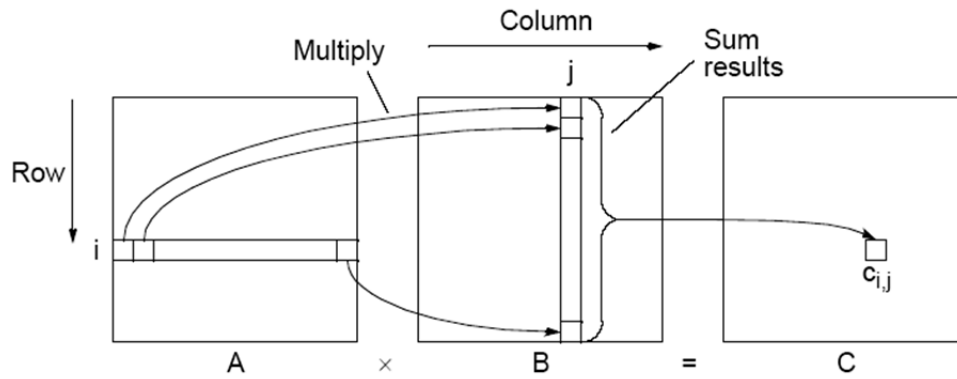


The screenshot shows a terminal window titled "ParallelProg-32-FINAL [Running] - Oracle VM VirtualBox" with a menu bar (File, Machine, View, Input, Devices, Help) and a title bar. The desktop environment is Ubuntu. The terminal prompt is "abw@abw-VirtualBox:~/ParallelProg/Suzaku". The user has executed the command "mpirun -n 3 test_workpool2". The output shows a sequence of MPI tasks (Diffuse, Compute, Gather) being sent and received across three processes (taskID 0, 1, 2, 3). Each task includes a "Hello World" message and a set of coordinates (x, y, z[0][0], z[1][1]). The output is as follows:

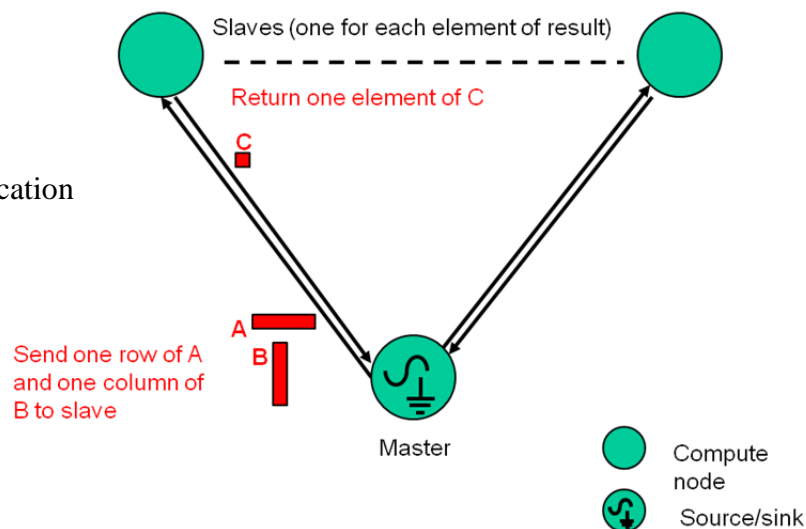
```
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpirun -n 3 test_workpool2
number of tasks = 4
Diffuse Task sent: taskID = 0, w = Hello World, x = 1234, y = 5678.00, z[0][0] = 357.00, z[1][1] = 246.00
Diffuse Task sent: taskID = 1, w = Hello World, x = 1235, y = 5679.00, z[0][0] = 357.00, z[1][1] = 246.00
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$
```

2. Matrix Multiplication

A simple (but very inefficient way) to do matrix multiplication as a workpool is to send one row of A and one column of B as one task to a slave to compute one element of the result as shown below:



Matrix multiplication
workpool



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

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

#define N 4 // size of matrices
#define T N * N // required Suzaku constant, number of tasks, max = INT_MAX - 1

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

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

void diffuse(int taskID) { // uses same approach as Seeds sample but inefficient copying arrays
```



```

int i;
int a, b;
double rowA[N],colB[N];

a = taskID / N;      // row
b = taskID % N;     // column
for (i = 0; i < N; i++) {
    rowA[i] = A[a][i]; // copy row of A.
                        // Strictly do not need to do this as can specify one row in SZ_Put("rowA",A[a]);
    colB[i] = B[i][b]; // copy one column of B into output
}

SZ_Put("rowA",rowA);
SZ_Put("colB",colB);
return;
}

void compute(int taskID) {
    int i;
    double out;
    double rowA[N],colB[N];

    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 out;

    SZ_Get("out",&out);
    a = taskID / N;
    b = taskID % N;
    C[a][b]= out;

    return;
}

// additional routine

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[]) {
    int i,j,k; // All variables declared here are in every process

```

```

int p;                // number of processes, set by SZ_Init()
double sum;
double time1, time2; // for timing in master

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

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

// sequential matrix multiplication, answer in D
time1 = SZ_Wtime(); //start time measurement
for (i = 0; i < N; i++) {
    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;
    }
}
time2 = SZ_Wtime(); //end time measurement

printf("Time of sequential computation: %f seconds\n", time2-time1);

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

    SZ_Workpool2(init,diffuse,compute,gather); // workpool matrix multiplication,answer in C

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

printf("Time of parallel computation: %f seconds\n", time2-time1);

printf("Array A");
print_array(A);
printf("Array B");
print_array(B);
printf("Array C");
print_array(C);

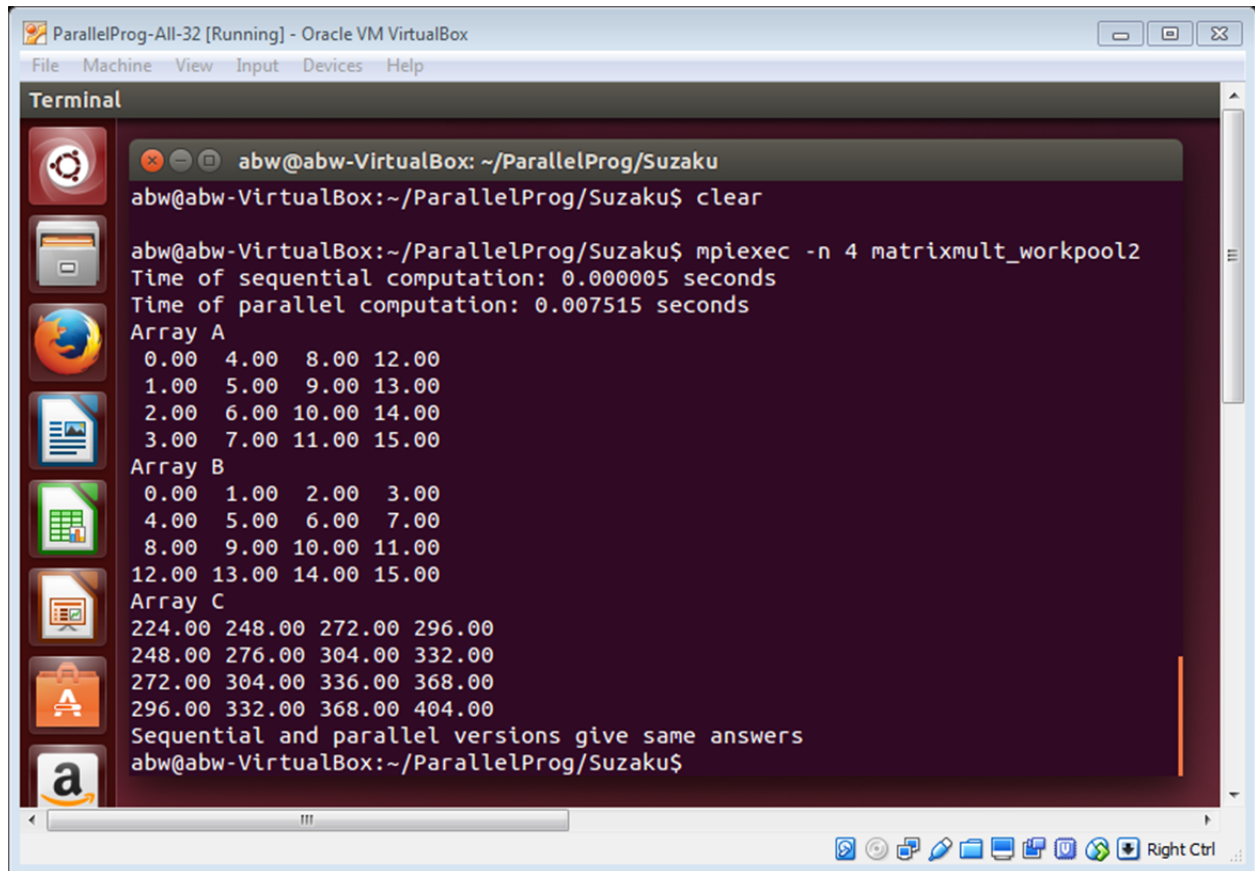
// check sequential and parallel versions give same answers
int error = 0;
for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
        if (C[i][j] != D[i][j]) error = -1;
    }
}
if (error == -1) printf("ERROR, sequential and parallel versions give different answers\n");
else printf("Sequential and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

SZ_Finalize();

return 0;
}

```

Sample output:



```
ParallelProg-All-32 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help

Terminal

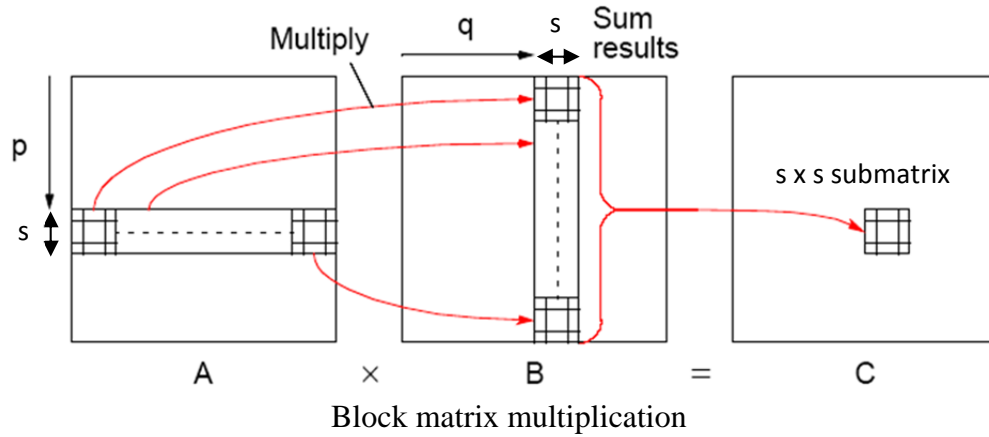
abw@abw-VirtualBox: ~/ParallelProg/Suzaku
abw@abw-VirtualBox:~/ParallelProg/Suzaku$ clear

abw@abw-VirtualBox:~/ParallelProg/Suzaku$ mpirun -n 4 matrixmult_workpool2
Time of sequential computation: 0.000005 seconds
Time of parallel computation: 0.007515 seconds
Array A
0.00 4.00 8.00 12.00
1.00 5.00 9.00 13.00
2.00 6.00 10.00 14.00
3.00 7.00 11.00 15.00
Array B
0.00 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
Array C
224.00 248.00 272.00 296.00
248.00 276.00 304.00 332.00
272.00 304.00 336.00 368.00
296.00 332.00 368.00 404.00
Sequential and parallel versions give same answers
abw@abw-VirtualBox:~/ParallelProg/Suzaku$
```

Notice parallel version is significantly slower than the sequential version with such small 4 x 4 arrays.

3. Block Matrix Multiplication

A better matrix multiplication method than the previous method is to use *block matrix multiplication algorithm* as shown below:



Each slave is given s rows and s columns. Pairs of $s \times s$ submatrices are multiplied and the results added together to produce an $s \times s$ submatrix answer.

This is set as a student assignment to implement and not posted here. It is quite easy to modify `matrixmult_workpool2.c`