Assignment 6
Suzaku Workpool Version 2 Programming Assignment

B. Wilkinson Modification date March 27, 2016

Overview

This assignment is new for Spring 2016 and replaces the previous short assignment on the Seeds framework. 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 that version 1 but is more powerful and more elegant to use. For this assignment, you will use your own computer only. Again we are interested in the ease of use rather than raw execution speed.

Workpool version 2 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: **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).

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).

Workpool routines

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

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
***compute**	Function pointer to gather function

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.

Preliminaries

The files needed for this assignment are:

- **suzaku.h**
- **suzaku.c**
- **test_workpool2.c**
- **matrixmult_workpool2.c**
- **makefile** (optional but useful)

which you will need to download from the course home at: "Parallel Programming Software" under "Suzaku." Place the files in the directory **ParallelProg/Suzaku**.

Part 1 Tutorial on the Suzaku workpool version 2 (30%)

Task 1 Compile and execute a simple workpool test program

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

```
#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;
}
```

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.)

Compile and execute this program with three processes.
Save a screenshot of the output. Make sure you understand the program and its output.

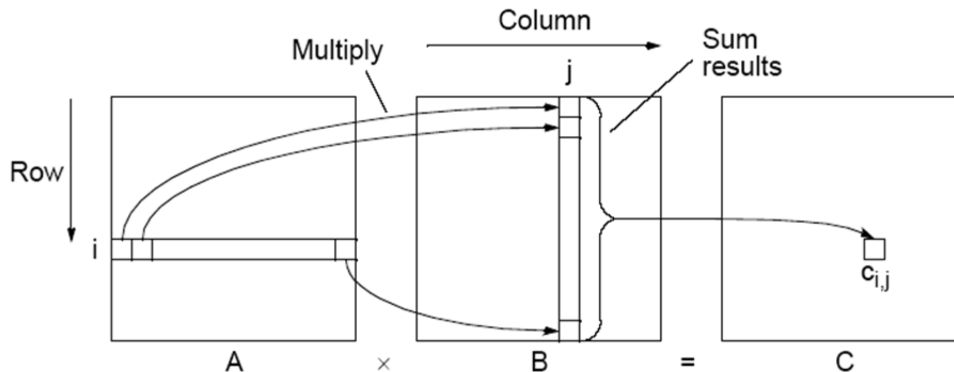
What to submit for Task 1

Your submission document should include the following:

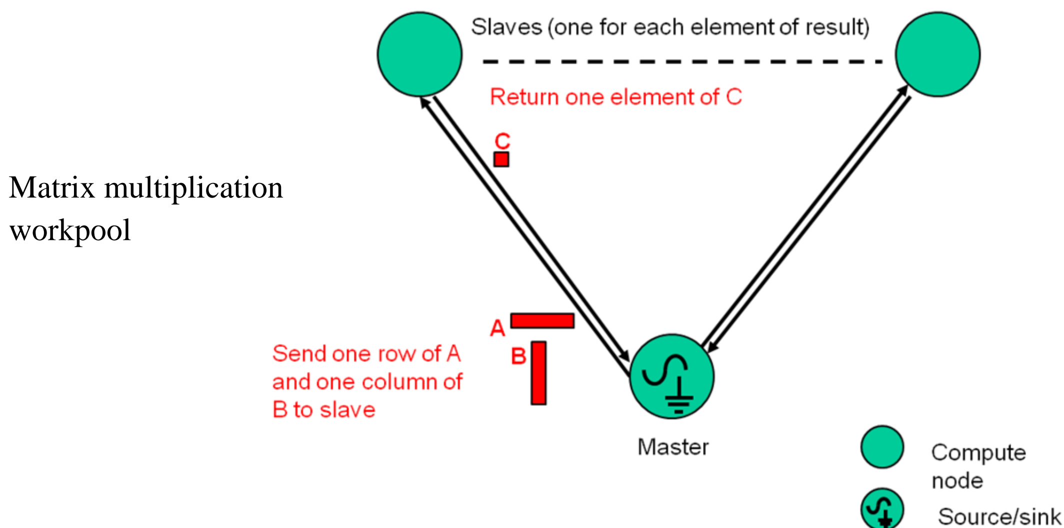
1. Screenshot of the **test_workpool2.c** executing with three processes.

Task 2 Matrix Multiplication using workpool version 2

Multiplication of two matrices, **A** and **B**, to produce matrix **C** is shown below:



In the workpool implementation, one slave computes one $c_{i,j}$ element of the result as shown below:



(Note rows of ***A*** *and columns of* ***B*** *are send to slaves;* ***B*** *is not broadcast.)*

A program implementing this matrix multiplication 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 as can do SZ_Put("rowA",A[a]);
                             // but will be needed in block multiplication
      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;
}
```

Compile and execute this program with three processes.
Try increasing the size of the matrices (e.g. 8 x 8, 16 x 16, etc) and report on your findings.
Save a screenshot of the output. Make sure you understand the program and its output.

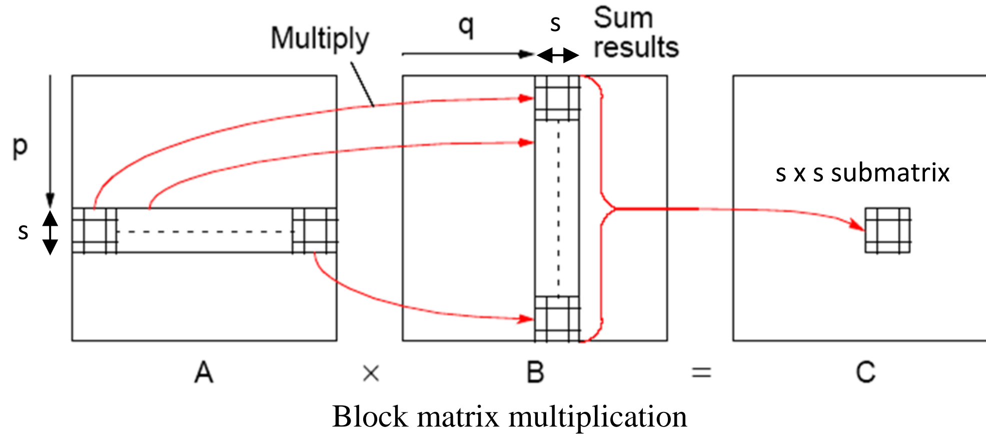
What to submit for Task 2

Your submission document should include the following:

1. Screenshot of the **matrixmult_workpool2.c** executing with four processes.
2. Report on what happens when you increase the size of the matrices and why.

Part 2 Writing Your Own Code – Block Matrix Multiplication (70%)

Modify the sample matrix multiplication program to multiple two *N* x *N* matrices using the *block matrix multiplication algorithm* shown below:



Block matrix multiplication

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

Choose *N* = 8 and *s* = 2. With *s* = 2, there are 16 submatrices and 16 tasks (*T* = *16*). Test your program with four processes. Make sure the parallel code gives the same answers as the sequential code.

It is extremely important that you implement the algorithm as described. It is not acceptable to use the matrixmult_workpool2.c code simply with 8 x 8 array size.

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.

What to submit from this part

Your submission document should include the following:

1. A code listing of your program for block matrix multiplication, sufficiently commented so that we can understand the program
2. Screenshot of your program executing with four processes.
3. Conclusions on using Suzaku workpool version 2 for block matrix multiplication. What are the advantages over using the regular matrix multiplication algorithm in Part 1? (5 points)
4. Give a brief evaluation comparing and contrasting using Suzaku with MPI. Describe your experiences and opinions, and give any suggestions for improvement. (5 points)

Assignment Report Preparation and Grading

Produce a report that shows that you successfully followed the instructions and performs all tasks by taking screenshots and include these screenshots in the document.

Every task and subtask specified will be allocated a score so make sure you clearly identify each part/task you did. **Include code, not as screen shots of the listings but complete properly documented code listing.** The programs are often too long to see in a single screenshot and also not easy to read if small. Using multiple screenshots is not option for code. Copy/paste the code into the Word document or whatever you are using to create the report.

You will lose 10 points if you submit code as screenshots.

Assignment Submission

Convert your report into a ***single pdf*** document and submit the pdf file on Moodle by the due date as described on the course home page. It is extremely important you submit only a pdf file.

You will lose 10 points if you submit in any other format (e.g. Word, OpenOffice, ...). Submitting a zip with multiple files is especially irritating.