SUZAKU Pattern Programming Framework Specification

4 - Workpool Version 3

B. Wilkinson March 17, 2016

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* and fro differentiation use the term *static workpool* where the number of tasks in the task queue is fixed.

4.1 Workpool Routines

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.

4.2 Signtaure of Suzaku Workpool 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. *This allows version 2 application code to execute with SZ_workpool3() without any change to the application code.*

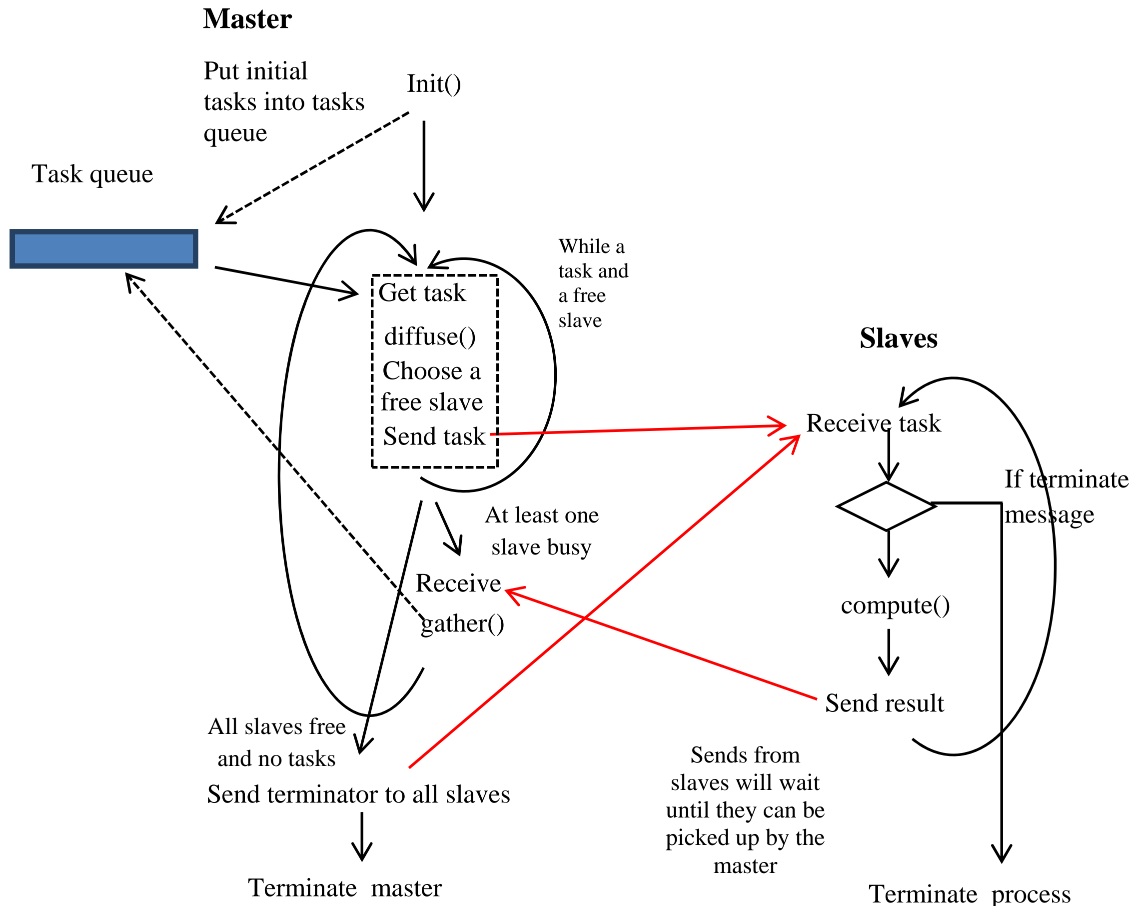
4.3 Compilation and Execution

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.

4.4 Implementation Details

The workpool algorithm implemented for Version 3 is shown overleaf: 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



Dynamic workpool algorithm

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.

Sample programs

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

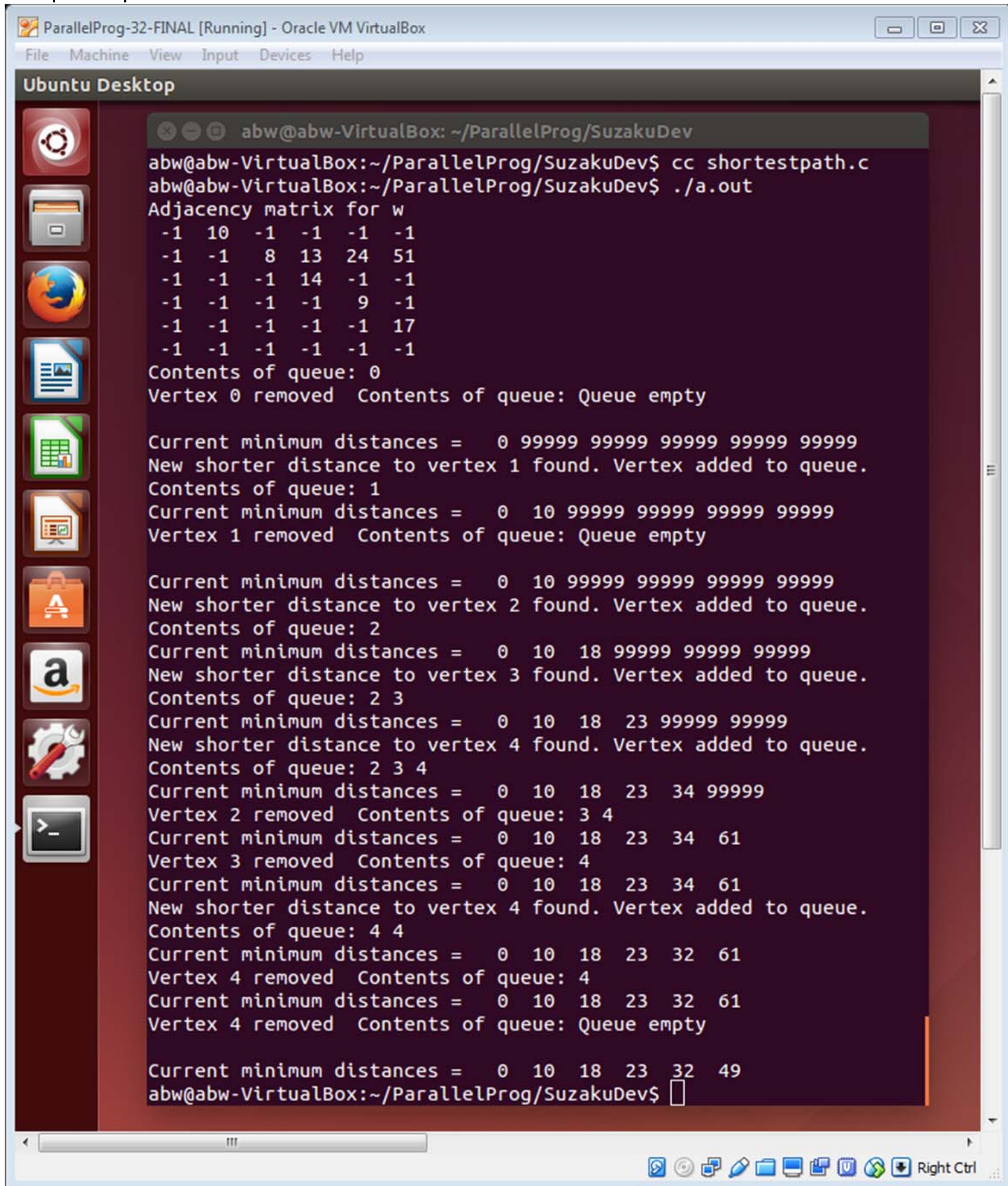
From page 214, *Parallel Programming Techniques and Applications*, 2nd ed. by B. Wilkinson, Prentice Hall 2005.

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



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],dist[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

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$
```