MPI parallel bucksort algorithm report

- **Introduction**

The computer industry is undergoing, if not another revolution, certainly a strong raising-up in parallel programing. Researcher found that the production of parallel compilers that automaticlly paralleslise many programers have provided painless route to parallelization (Advances in compters, 2000). In addtion, to the development of the low written parallel applications, which is ported across for different platforms.

The major chip manufacturers have, for the time being at least, given up trying to make processors run faster. Instead, manufacturers are turning to "multicore" architectures, in which multiple processors (cores) communicate directly through shared hardware caches (Advances IT,2001).

Multiprocessor chips make computing more effective by exploiting **parallelism** such that:

- **Harnessing multiple processors to work on a single task.**
- **Helps the ablity to tackle large problems that today's single processor.**
- **Helps to split memory useage between sperated processor.**
- **Helps the ability to increase the speed of computers even when the size of processors become smaler.**

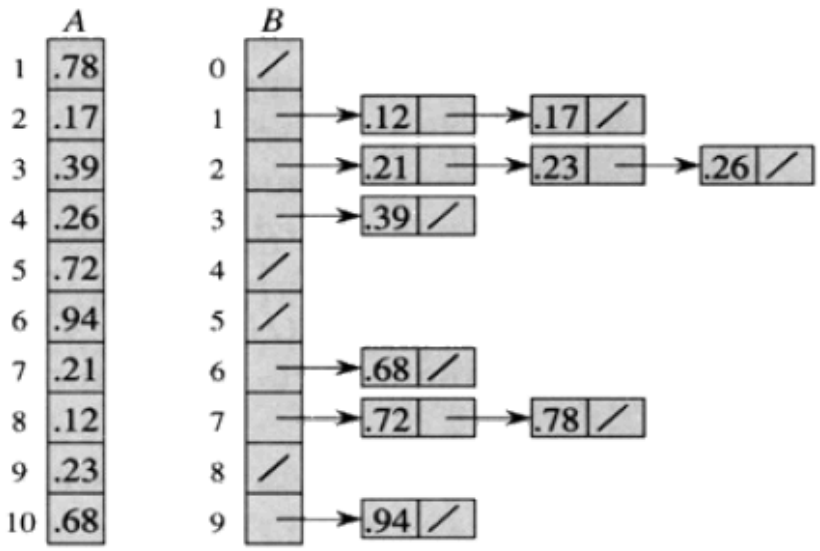
The following report will dicuess the design of the parrallel bucksort algorithm which is written in C using MPI libuary communcation between processes. Firstly, a breif meaning of bucket sort algorithm is given, including a example of how it works. Secoundly, two tests of implmlitions which uses the C program to disrubted the result. The first test uses only one processor and the secound test uses more than one processors. Lastly, a conclution of the report, list of references and the C code is provided at the end.

- **What is Bucket Sort Algorithm?**

Bucket sort is a sorting algorithm that works by partitioning an array into a number of buckets. Each bucket is then sorted individually.The idea of Bucket sort is to divide the interval (0, 1) into n equal-sized subintervals, or buckets, and then distribute the n input numbers into the buckets. Since the inputs are uniformly distributed over (0, 1), we don't expect many numbers to fall into each bucket. To produce the output, simply sort the numbers in each bucket and then go through the bucket in order, listing the elements in each (introdcution to algorithm, 2009). Bucket sort algorithm uses $O(n^2)$ in best case and O(n) in average case.

Example

Given input array A[1..10]. The array B[0..9] of sorted lists or buckets after line 5. Bucket i holds values in the interval [*i*/10, (*i*+1)/10]. The sorted output consists of a concatenation in order of the lists first B[0] then B[1]thenB[2]... and the last one is B[9].



The following figure show how bucket sort algorithm works.

- **Implementation :**

  Test 1 only uses one processors.

  Enviroment :

  The bucket Sort algoritm was implemented and tested with the following tools and hardware:

  - MPI type: Open Mpi version 1.4.2
  - Machine: Macbook Pro 15 inch

  Following test used for one machine which used the bucket sort algorithm:

```
Abdulazizs-MBP:mpi assignment AbdulazizJamal$ /Users/AbdulazizJamal/Downloads/mpi\ assignment/a.out 10

N Procs = 1  Array size = 10

Full list:  0.135021 0.295063 0.123345 0.0647726 0.633516 0.511958 0.469904 0.682408 0.235712 0.619579
Final sorted List:  0.0647726 0.123345 0.135021 0.235712 0.295063 0.469904 0.511958 0.619579 0.633516 0.682408
Total time: 0.000024 secs
Abdulazizs-MBP:mpi assignment AbdulazizJamal$
```

  Test 2 uses more than one processors.

  Enviroment:

  The bucket Sort algoritm was implemented and tested with the following tools and hardware:

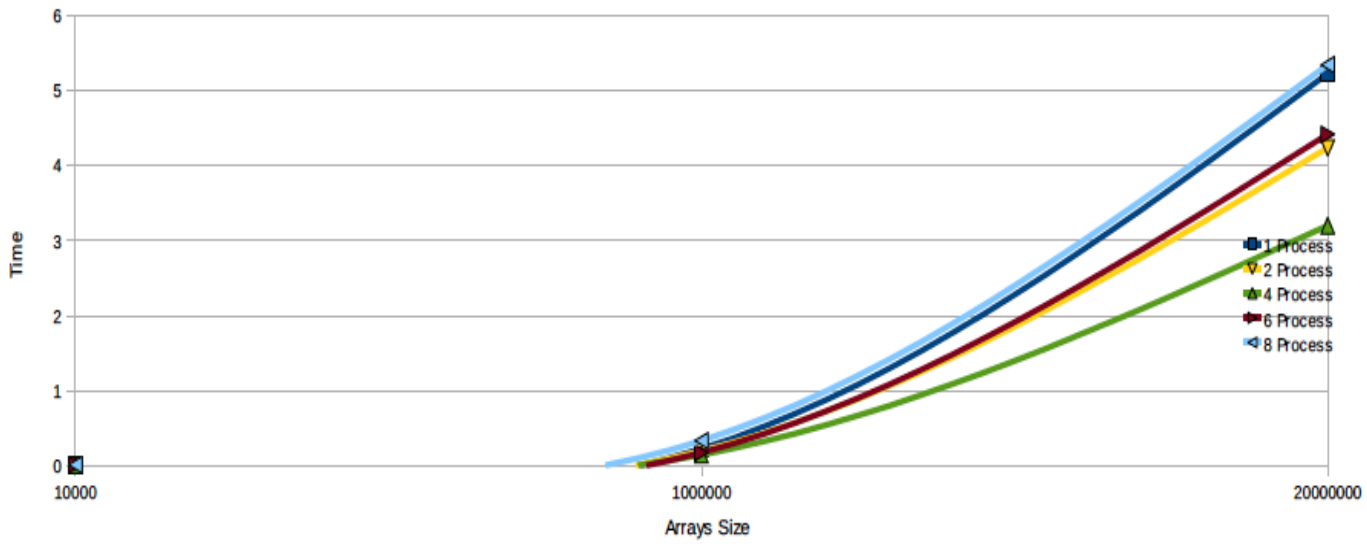
  - MPI Type: MPICH

- Machines in DEC10: cslin032, cslin033,cslin034, cslin035, cslin036, cslin037, cslin038, cslin039.

- **Test cases and results**

  The following table shown the result (time used to sort the array) for differents numbers of processors and array sizes values.

	N = 10000	N = 1.000.000	N= 20.000.000
P = 1	• 0,0017 secs • 0,0020 secs • 0,0019 secs	• 0,222 secs • 0,220 secs • 0,226 secs	• 5,25 secs • 5,23 secs • 5,21 secs
P = 2	• 0,0041 secs • 0,0031 secs • 0,0020 secs	• 0,146 secs • 0,191 secs • 0,192 secs	• 4,32 secs • 4,24 secs • 4,41 secs
P = 4	• 0,0035 secs • 0,0040 secs • 0,0019 secs	• 0,145 secs • 0,140 secs • 0,141 secs	• 3,14 secs • 3,2 secs • 3,5 secs
P = 6	• 0,0073 secs • 0,0057 secs • 0,0062 secs	• 0,170 secs • 0,175 secs • 0,248 secs	• 4,23 secs • 4,42 secs • 4,42 secs
P = 8	• 0,0073 secs • 0,0073 secs • 0,0055 secs	• 0,305 secs • 0,333 secs • 0,306 secs	• 5,14 secs • 5,35 secs • 5,52 secs



- **Conlustion**

  On conlustion, this report disscsed about the test of the parrallel bucksort algorith. As we can see in the graph above, for short arrays the parallelism does not increase the performance of the algorithm. That's because the time employed to synchronize process is greater than the time employed to sort the array. For long arrays, the parallelism show a significant increase on performance, and particular on p = 4 (number of processors of the target computer).

- **References**

Advances in the Computer Simulations of Liquid Crystalsby Paolo Pasini, Claudio Zannoni textbook - (2000)
Advanced Information Technology in Education textbook – 2002
Introduction to Algorithms - 2009

- **C code (used for the tests)**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>               /* Used to seed the random number generator */
#include <mpi.h>



/*
        Serial quicksort in-place. You should call the routine "serialQuicksort()" (below) as:

        serialQuickSort( list, start, end );

        with

        float *list;                    List to be sorted
        int start;                              Start index (initially 0; becomes non-zero during recursion)
        int end;                        End index (i.e. the list size, if start=0).

*/
int serialPartition( float *list, int start, int end, int pivotIndex )
{
        /* Scratch variables */
        float temp;

        /* Get the pivot value */
        float pivotValue = list[pivotIndex];

        /* Move pivot to the end of the list (or list segment); swap with the value already there. */
        list[pivotIndex] = list[end-1];
        list[end-1]      = pivotValue;

        /* Sort into sublists using swaps; get all values below the pivot to the start of the list */
```

```
	int storeIndex = start, i;				/* storeIndex is where the values less than the pivot go */
	for( i=start; i<end-1; i++ )		/* Don't loop over the pivot (currently at end-1); will handle that just before returning */
	if( list[i] <= pivotValue )
	{
		/* Swap value with this element and that at the current storeIndex */
		temp = list[i]; list[i] = list[storeIndex]; list[storeIndex] = temp;

		/* Increment the store index; will point to one past the pivot index at the end of this loop */
	storeIndex++;			/* Could do this at the end of the previous instruction */
	}

	/* Move pivot to its final place */
	temp = list[storeIndex]; list[storeIndex] = list[end-1]; list[end-1] = temp;

	return storeIndex;
}



/* Performs the quicksort (partitioning and recursion) */
void serialQuicksort( float *list, int start, int end )
{
	if( end > start+1 )
	{
	int pivotIndex = start;				/* Choice of pivot index */

	/* Partition and return list of pivot point */
	int finalPivotIndex = serialPartition( list, start, end, pivotIndex );

	/* Recursion on sublist smaller than the pivot (including the pivot value itself) ... */
	serialQuicksort( list, start, finalPivotIndex );

	/* .. and greater than the pivot value (not including the pivot) */
	serialQuicksort( list, finalPivotIndex+1, end );
	}
	/* else ... only had one element anyway; no need to do anything */
}
```

```
void displayFullList( float *list, int n )
{

        /* Do not display large lists */
        if( n>100 )
        {
        printf( "Not displaying 'full list' - n too large.\n" );
        return;
        }

        int i;
        for( i=0; i<n; i++ ) printf( " %g", list[i] );
        printf( "\n" );
}


/*
        Main
*/
int main( int argc, char **argv )
{

        int numprocs, rank, i, n, rc;
        double start_time, end_time;

        /* Replace these lines with the corresponding MPI routines */
        rc = MPI_Init(&argc,&argv);
        if (rc != MPI_SUCCESS){
        printf ("Error starting MPI program. Terminating.\n");
        MPI_Abort(MPI_COMM_WORLD, rc);
        }

        MPI_Comm_size (MPI_COMM_WORLD,&numprocs);
        MPI_Comm_rank (MPI_COMM_WORLD,&rank);


        /* Get n from the command line. Note that you should always finalise MPI before quitting. */
        if( argc != 2 ){
        if( rank==0 ) printf( "Need one argument (= the list size n).\n" );
        return EXIT_FAILURE;
```

```
	}

	n = atoi( argv[1] );

	if( n<=0 ){
	if( rank==0 ) printf( "List size must be >0.\n" );
	return EXIT_FAILURE;
	}

	if(rank == 0)
	printf( "\nN Procs = %d  Array size = %d\n\n",numprocs,n);


	start_time = MPI_Wtime();


	float *globalArray = NULL;		/* Initial unsorted list, and final sorted list; only on rank==0 */
	float *short_bucket[numprocs];



	if( rank==0 ){
	MPI_Status status;
	int j;
	int bucket_sizes[numprocs];			// Size of each bucket
	srand( time(NULL) );				// Seed the random number generator to the system time

	globalArray = (float*) malloc( sizeof(float)*n );
	float bucket_lim = 1.0 / numprocs;		//Pivot

	/* Create an array of size n and fill it with random numbers*/
	for( i=0; i<n; i++ )
		globalArray[i] = 1.0*rand() / RAND_MAX;

	printf("Full list: ");
	displayFullList(globalArray,n);

	/*Alloc memory for buckets and initialize the size on 0*/
	for(i = 0; i < numprocs; i++){
```

```
        short_bucket[i] = (float*) malloc( sizeof(float)* n );
        bucket_sizes[i] = 0;
}

/* Partitionate the global array into the bukets*/
for( i=0; i<n; i++ ){
        for(j=0; j<numprocs; j++){
                if((globalArray[i] > (bucket_lim * j)) && (globalArray[i] <= (bucket_lim * (j+1)))){
                        short_bucket[j][bucket_sizes[j]] = globalArray[i];
                        bucket_sizes[j]++;
                }
        }
}

/*Send buckets to each process*/
for(i=1; i<numprocs; i++){
        MPI_Send( &bucket_sizes[i], 1, MPI_INT, i, 0, MPI_COMM_WORLD );
        MPI_Send( short_bucket[i], bucket_sizes[i], MPI_FLOAT, i, 0, MPI_COMM_WORLD );
}

/*Sort bucket 0*/
serialQuicksort(short_bucket[0],0,bucket_sizes[0]);


/*Receive Sorted buckets from each process*/
for(i=1; i<numprocs; i++){
  MPI_Recv(short_bucket[i], bucket_sizes[i], MPI_FLOAT, i, 0, MPI_COMM_WORLD, &status );
}


/*Concatenate sorted buckets*/
int index = 0;
for(i=0; i<numprocs; i++){
        for(j=0; j< bucket_sizes[i]; j++){
                globalArray[index++] = short_bucket[i][j];

        }
}

printf("Final sorted List: ");
```

```
        displayFullList(globalArray,n);

        }
        /*Rank != 0*/
        else{
        int size;

        MPI_Status status;

        /*Receive the size of bucket*/
        MPI_Recv( &size, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status );

        /*Alloc memory and receive the bucket*/
        float *bucket = (float*) malloc( sizeof(float)*size );
        MPI_Recv( bucket, size, MPI_FLOAT, 0, 0, MPI_COMM_WORLD, &status );


        printf("Procces %d:  bucket size %d.  Elemets: ",rank,size);
        displayFullList(bucket,size);

        serialQuicksort(bucket,0,size);
        printf("Procces %d:  Sorted Elemets: ",rank);
        displayFullList(bucket,size);

        MPI_Send( bucket, size, MPI_FLOAT, 0, 0, MPI_COMM_WORLD);
        }


        end_time = MPI_Wtime();

        if(rank == 0)
        printf("Total time: %f secs\n",end_time - start_time);


        MPI_Finalize();

        return 0;

}
```