**FAST National University of Computer and Emerging Sciences**



**Research Topic:** To what extent can parallelism improve time efficiency of Bubble Sort Algorithm over a large dataset?

**Instructor:** Nadeem Kafi

**Group Members:**

1. Aahil Halani (20K-0308)
2. Ali Ahmed (20K-0140)
3. Faizan Hakim (20K-0290)

**Date: 17th November, 2022**

**Table of Contents**

1. Introduction -------------------------------------------------------------------------- 3
2. Methodology ------------------------------------------------------------------------- 3
3. System Specifications --------------------------------------------------------------- 4
4. Program Codes ----------------------------------------------------------------------- 4
5. Results ------------------------------------------------------------------------------- 9
6. Limitations -------------------------------------------------------------------------- 12
7. Analysis ---------------------------------------------------------------------------- 12
8. Evaluation -------------------------------------------------------------------------- 12
9. Conclusion -------------------------------------------------------------------------- 12
10. **Introduction**

Bubble sort algorithm is one of the most commonly used comparative sort algorithm that can be used to sort a collection of data elements in either ascending or descending order. It has a space complexity of O (1), making it a viable option when memory is limited. However, in time bound situations, this algorithm could perform poorly over large data set as average time complexity of the algorithm is calculated as Ꝋ (). This implies that the algorithm has a tendency to consume large amount of CPU resources which in result degrades system’s overall performance. Hence, to overcome this effect, parallelism techniques could be utilized to improve the time and overall CPU utilization of the algorithm. The techniques of parallelism that are going to be utilized in this project lie in the domain of Open-MP and MPI libraries.

Open-MP is a high-level approach to parallelism based on threads sharing a memory space. Tasks and relevant data sets are distributed amongst implicitly created threads with in a single system and results from each thread can be communicated within threads using a common shared memory. Threads can be tracked using unique thread ids and inclusion of many highly functional, user-friendly clauses make this approach easy to implement.

MPI on the other hand, is a message passing based approach relying on a distributed memory scheme. Tasks and memory sets are distributed to number of processes working in a single MPI communication network sometimes referred MPI\_COMM\_WORLD. Each process within this communication world has an ability to send, receive, or gather data elements from all other processes lying in the same communication world by utilization many message passing calls.

Both approaches have a common technique to distribute independent tasks and gather results from each distribution to combine into a single solution, saving time and CPU resources. Therefore, this study aims to investigate the highest gain in performance that can be achieved by using either of the two parallelism models. This information can hopefully come in hand for developers when designing and implementing a real time, time bound systems.

1. **Methodology**
2. A data set containing n random values would be created. (n size may vary to 1K to 100K).
3. Bubble sort algorithm will be tailored and implemented on data set using each parallelism module (Open-MP and MPI) separately.
4. Performance of each module will be recorded upon each task completion. The time stamp with number of thread or process used will be stored in a file; separate files for Open-MP and MPI.
5. Comparison in performance will be made according to time taken to complete the task against varying number of process and threads.
6. **System Specifications**

* Intel Core i7 – 6600
* Processor Base Speed 2.4 GHz
* Number of Cores – 2
* RAM size – 4 GB

1. **Program Codes**
2. **Open MP**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 1000000

int main ()

{

int A[SIZE];

int A2[SIZE];

int x;

printf("Enter the number of threads you want to create: "); scanf("%d", &x);

FILE\* fptr;

fptr = fopen("parallel.txt", "r");

if (fptr == NULL)

{

printf("ERROR");

exit(1);

}

int i = 0, j=0;

int count = 0;

while (fscanf(fptr, "%d", &A[i]) == 1)

{

A2[i] = A[i];

count++;

i++;

}

fclose(fptr);

int N = count;

double start,end;

start=omp\_get\_wtime();

#pragma omp parallel num\_threads(x)

{

for( i = 0; i < N; i++ )

{

#pragma omp parallel for shared(A,i,N)

for( j = 0; j < N-i-1; j++ )

{

if( A[ j ] > A[ j+1 ] )

{

int temp = A[j];

A[j] = A[j+1];

A[j+1] = temp;

}

}

}

}

end=omp\_get\_wtime();

printf("\nTime Parallel= %f\n",(end-start));

FILE \* file = NULL;

FILE \* file1 = NULL;

file = fopen("result", "w");

file1 = fopen("Timelog","a");

fprintf(file,"Size: ");

fprintf(file, "%d\n", SIZE); // assert (s == n)

for (i = 0; i < SIZE; i++)

fprintf(file, "%d\n", A[i]);

fclose(file);

fprintf(file1,"Size: ");

fprintf(file1, "%d\n", SIZE);

fprintf(file1,"Threads: ");

fprintf(file1,"%d\n",x);

fprintf(file1,"%f seconds\n\n",(end-start));

fclose(file1);

}

1. **MPI**

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

#include <stdlib.h>

#define N 1000000

/\* swap entries in array v at positions i and j; used by bubblesort \*/

static inline

void swap(int \* v, int i, int j)

{

int t = v[i];

v[i] = v[j];

v[j] = t;

}

/\* (bubble) sort array v; array is of length n \*/

void bubblesort(int \* v, int n)

{

int i, j;

for (i = n-2; i >= 0; i--)

for (j = 0; j <= i; j++)

if (v[j] > v[j+1])

swap(v, j, j+1);

}

/\* merge two sorted arrays v1, v2 of lengths n1, n2, respectively \*/

int \* merge(int \* v1, int n1, int \* v2, int n2)

{

int \* result = (int \*)malloc((n1 + n2) \* sizeof(int));

int i = 0;

int j = 0;

int k;

for (k = 0; k < n1 + n2; k++) {

if (i >= n1) {

result[k] = v2[j];

j++;

}

else if (j >= n2) {

result[k] = v1[i];

i++;

}

else if (v1[i] < v2[j]) { // indices in bounds as i < n1 && j < n2

result[k] = v1[i];

i++;

}

else { // v2[j] <= v1[i]

result[k] = v2[j];

j++;

}

}

return result;

}

int main(int argc, char \*\* argv)

{

int n;

int \* data = NULL;

int c, s;

int \* chunk;

int o;

int \* other;

int step;

int p, id;

MPI\_Status status;

double elapsed\_time;

FILE \* file = NULL;

FILE \* file1 = NULL;

int i;

MPI\_Init(&argc, &argv);

MPI\_Comm\_size(MPI\_COMM\_WORLD, &p);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id);

if (id == 0) {

// read size of data

file = fopen("parallel.txt", "r");

// compute chunk size

n = N;

c = n/p; if (n%p) c++;

// read data from file

data = (int \*)malloc(p\*c \* sizeof(int));

for (i = 0; i < n; i++)

fscanf(file, "%d", &(data[i]));

fclose(file);

// pad data with 0 -- doesn't matter

for (i = n; i < p\*c; i++)

data[i] = 0;

}

// start the timer

MPI\_Barrier(MPI\_COMM\_WORLD);

elapsed\_time = - MPI\_Wtime();

// broadcast size

MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

// compute chunk size

c = n/p; if (n%p) c++;

// scatter data

chunk = (int \*)malloc(c \* sizeof(int));

MPI\_Scatter(data, c, MPI\_INT, chunk, c, MPI\_INT, 0, MPI\_COMM\_WORLD);

free(data);

data = NULL;

// compute size of own chunk and sort it

s = (n >= c \* (id+1)) ? c : n - c \* id;

bubblesort(chunk, s);

// up to log\_2 p merge steps

for (step = 1; step < p; step = 2\*step) {

if (id % (2\*step)!=0) {

// id is no multiple of 2\*step: send chunk to id-step and exit loop

MPI\_Send(chunk, s, MPI\_INT, id-step, 0, MPI\_COMM\_WORLD);

break;

}

// id is multiple of 2\*step: merge in chunk from id+step (if it exists)

if (id+step < p) {

// compute size of chunk to be received

o = (n >= c \* (id+2\*step)) ? c \* step : n - c \* (id+step);

// receive other chunk

other = (int \*)malloc(o \* sizeof(int));

MPI\_Recv(other, o, MPI\_INT, id+step, 0, MPI\_COMM\_WORLD, &status);

// merge and free memory

data = merge(chunk, s, other, o);

free(chunk);

free(other);

chunk = data;

s = s + o;

}

}

// stop the timer

elapsed\_time += MPI\_Wtime();

// write sorted data to out file and print out timer

if (id == 0) {

file = fopen("result", "w");

file1 = fopen("Timelog","a");

fprintf(file,"Size: ");

fprintf(file, "%d\n", s); // assert (s == n)

for (i = 0; i < s; i++)

fprintf(file, "%d\n", chunk[i]);

fclose(file);

printf("Bubblesort %d ints on %d procs: %f secs\n", n, p, elapsed\_time);

fprintf(file1,"Size: ");

fprintf(file1, "%d\n", s);

fprintf(file1,"Processors: ");

fprintf(file1,"%d\n",p);

fprintf(file1,"%f seconds\n\n",elapsed\_time);

fclose(file1);

}

MPI\_Finalize();

return 0;

}

1. **Results**

Number of Data Elements: 1 million (1000000)

* **Open-MP**

|  |  |
| --- | --- |
| Number of Threads | Time Taken (seconds) |
| 10 | 27.138156 |
| 20 | 26.990862 |
| 30 | 26.639837 |
| 40 | 26.608123 |
| 50 | 26.259490 |
| 60 | 26.181090 |
| 70 | 26.012241 |
| 80 | 25.927522 |
| 90 | 25.492561 |
| 100 | 24.845999 |

* **MPI**

|  |  |
| --- | --- |
| Number of Processes | Time Taken (seconds) |
| 10 | 348.125648 |
| 20 | 281.113952 |
| 30 | 198.526762 |
| 40 | 149.525025 |
| 50 | 117.165312 |
| 60 | 106.884673 |
| 70 | 113.203966 |
| 80 | 82.731768 |
| 90 | 90.058906 |
| 100 | 43.048867 |

1. **Limitations**

Tests have to be confined to element size of 1 million and number of threads/processes within the range of 10 to 100 as testing CPUs were unable to provide sufficient resources to threads/processes resulting in either a system deadlock, operating system crashing, or exponential runtimes.

1. **Analysis**

By increasing number of threads and processes, time taken by each process/thread to sort an array of 1 million elements, using bubble sort, decreases compared to the time taken to sort the array of same size and using same sorting technique serially. The graphs, relating to both Open-MP and MPI, indicates that as the number of threads/processes increases by the rate of 10, there is a negative change of time taken to execute the sorting technique. However, the graphs consist of several inflection points which indicates an increase in parallel run time due to an increase of threads/processes. Therefore, an optimal thread and process count for Open MP and MPI can be observed from the graph as 100 and 100 respectively; within test data.

1. **Evaluation**

The results of test runs using the algorithms stated in section 3 proves that using parallelism techniques there is a significant gain in performance in the system. As data has been divided into chunks and sent to respective threads/processes, considerable computation time has been reduced as computations are now being done concurrently in several distinct CPU cores. Much of the time taken by the program is now spent in data communication, hence, the system spends more time in communication rather than computation (fine grained decomposition) consequently increases the degree of parallelism and improving system’s speed. However, higher degree of parallelism may also make the system vulnerable to parallel overhead as many clock cycles of the CPU might not be utilized due to scheduling, initiation, termination, and communication between threads and processes; shown by inflection points on graphs in section 4. Therefore, a system might begin to take more time to run the algorithms if optimal number of threads or processes are not maintained.

1. **Conclusion**

Using parallel techniques of Open MP and MPI on bubble sort algorithm can increase its efficiency as computation as done in chunks and concurrently, resulting in lesser runtime and better utilization of system resources consequently increasing system’s overall performance.