Intro to distributed programming

Mohamad Ibrahim Cheikh

November 30, 2020

1 Introduction

Parallel computing refers to simulations in which multiple computational resources are implemented to solve a problem at the same time. Parallelism is usually achieved by dividing the problem into many smaller ones, in which each one is solved separately and also at the same time. Different forms of parallel programing has been developed like bit-level, instruction-level, data, and task parallelism.

The current report will analyze the performance of a distributed parallel execution model, that give the software the capability to control multiple different processes on the same or different machine which is *MPI*.

2 Experimental Configuration

The supercomputer on which we are going to conduct our performance analysis is Beocat, which is a high performance computer cluster at the Kansas State University [1]. The performance analysis will be performed on the Elves nodes of Beocat [2]. Table 1 gives a brief description of the nodes configuration.

Elve Nodes				
Nodes	1-56	57-72,77	73-76,78, 79	80-85
Processors	2x 8-Core Xeon E5-2690	2x 10-Core Xeon E5-2690 v2	2x 10-Core Xeon E5-2690 v2	2x 10-Core Xeon E5-2690v2
Ram	64GB	96GB	384GB	64GB
Hard Drive	1x 250GB 7,200 RPM SATA	1x 250GB 7,200 RPM SATA	1x 250GB 7,200 RPM SATA	1x 250GB 7,200 RPM SATA
NICs	4x Intel I350	4x Intel I350	4x Intel I350	4x Intel I350
10GbE and QDR Infiniband	MT27500 Family (ConnectX-3)	MT27500 Family (ConnectX-3)	MT27500 Family (ConnectX-3)	MT27500 Family (ConnectX-3)

Table 1: Elves node configuration taken from [2]

3 Code

This section will detail how the distributed programming code was implemented. Before discussing how the mpi-parallelized version of the code was implemented, we are going to walk over the steps that the code should implement:

- The first thing, the code will create an empty array for the keywords called `word`, and an array for the text that you want to look up for the keywords in called `line`.

- The second thing the code should do is read the **keyword.txt** file containing the list of keywords, and store them in a `word`, and read the **wiki_dump.txt** and store them in the array `line`.
- The third task is to look for the line at which the keyword in the `word` array occur and add it to a text.
- The final task is to write the out put on a text file.

Following the serial steps detailed above are shown in figure 1.

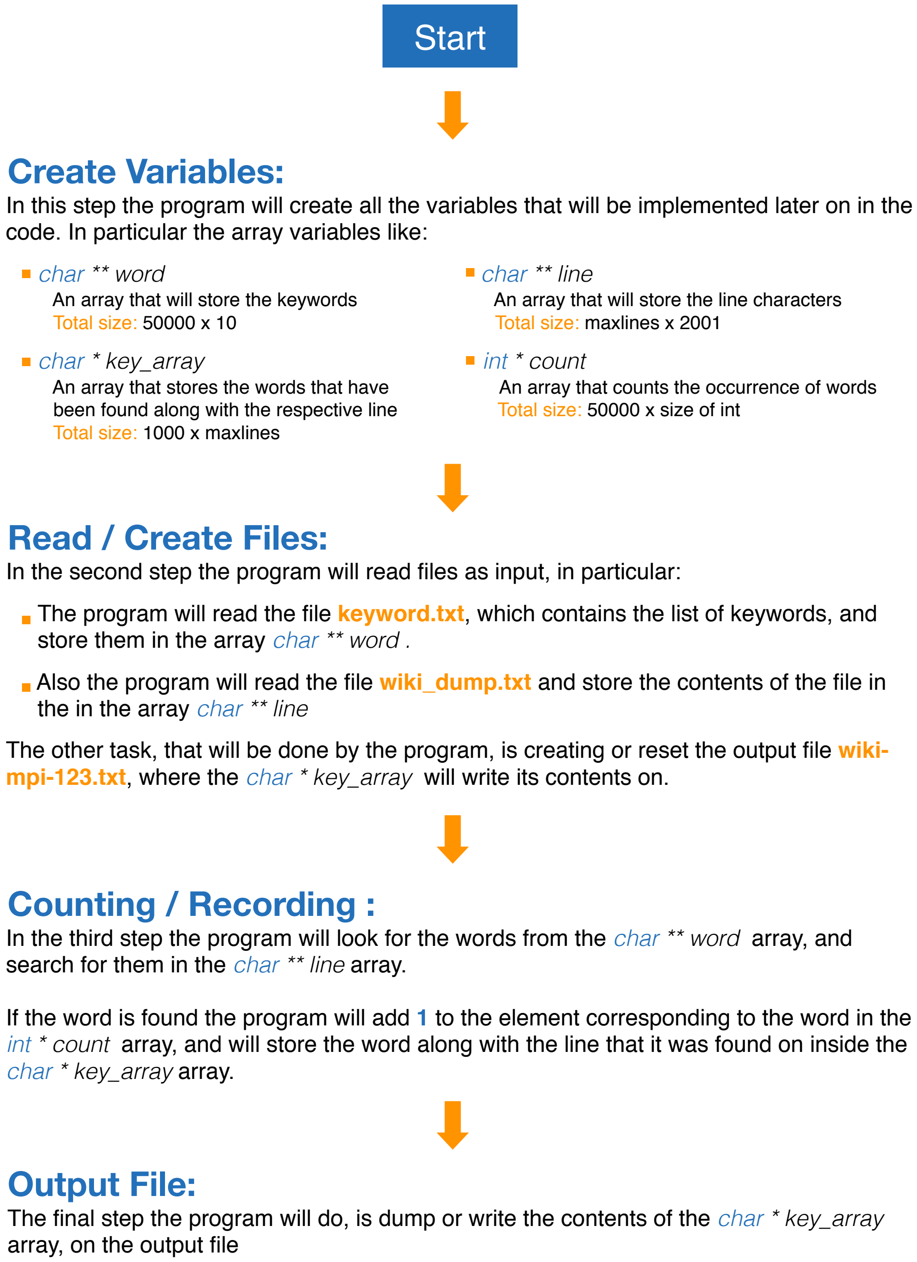


Figure 1: A list of tasks the program will do in serial

3.1 MPI Implementation

The current code was implemented in a way to minimize the amount of communication between the cores, to allow each core to work on its own without needing to depend on a another core. Figure 2 shows how the code was serialized for a 4 core simulation. From the figure it is clear that all tasks, starting from the initialization, to the reading, and ending with the counting are being done in parallel manner with no core depending on the other. The only time a processor has to wait is at the end when they need to write the output file.

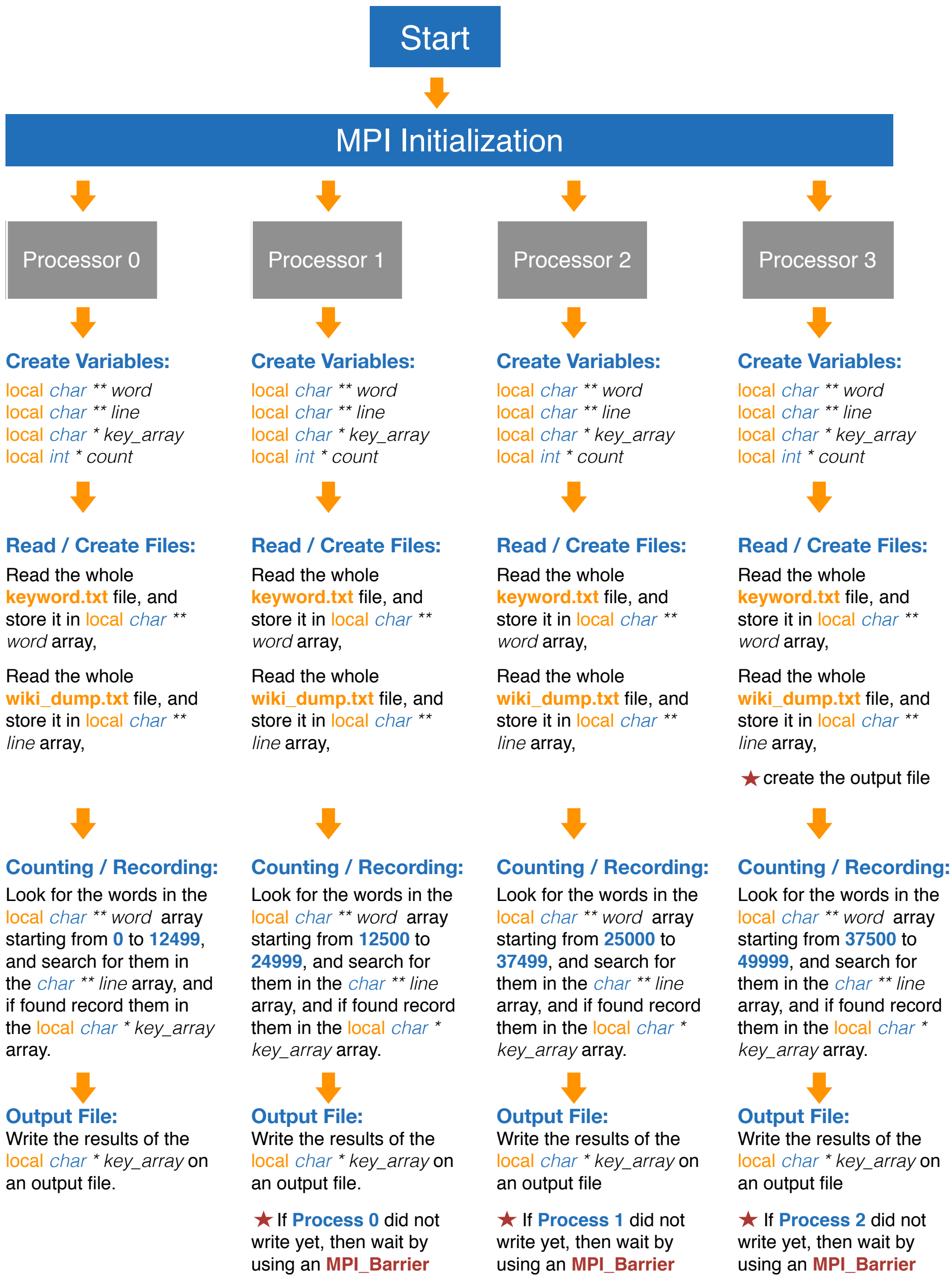


Figure 2: Parallel implementation of the serial code shown in figure 1

4 Results

For the current case the MPI implementation was divided in to two type, one type on will implement the cores on a *single machine*, and the other will divide the cores across *multiple machines*. The table below shows how many cores were run for each case.

	Number of Cores						
Type	1	2	4	8	16	32	64
Single Machine	X	X	X	X	X		
Multiple Machines			X	X	X	X	X

Table 2: Number of cores implemented for each parallel execution model

In addition to running on single or multiple machine, the MPI code was tested for different ranges of the the variable that controls the number of lines starting from $200,000$ and going up to $1,000,000$ with an increment of $200,000$. The results of the plot are shown in the figure 3.

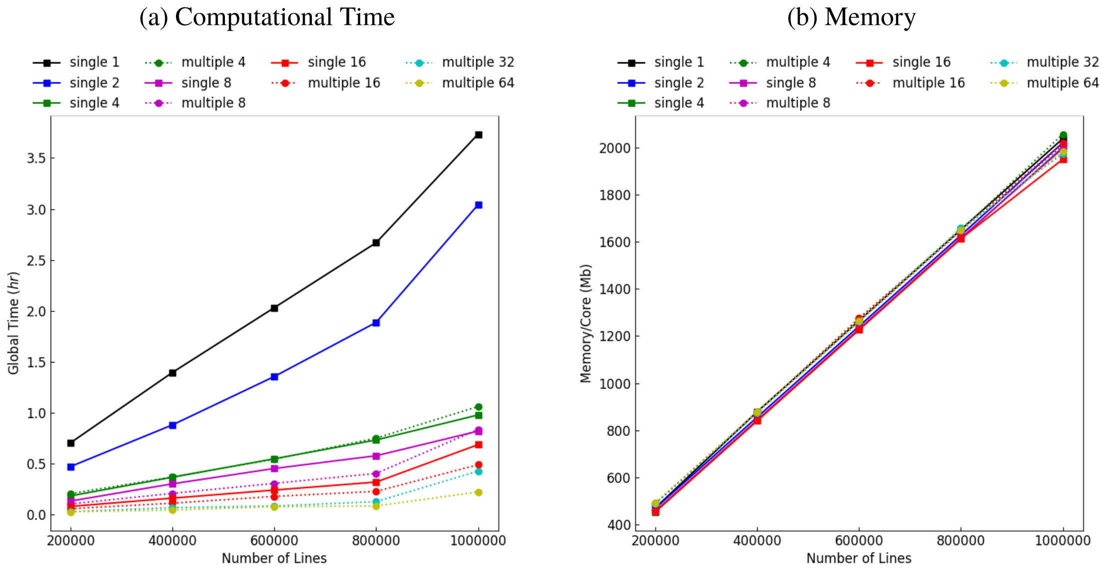
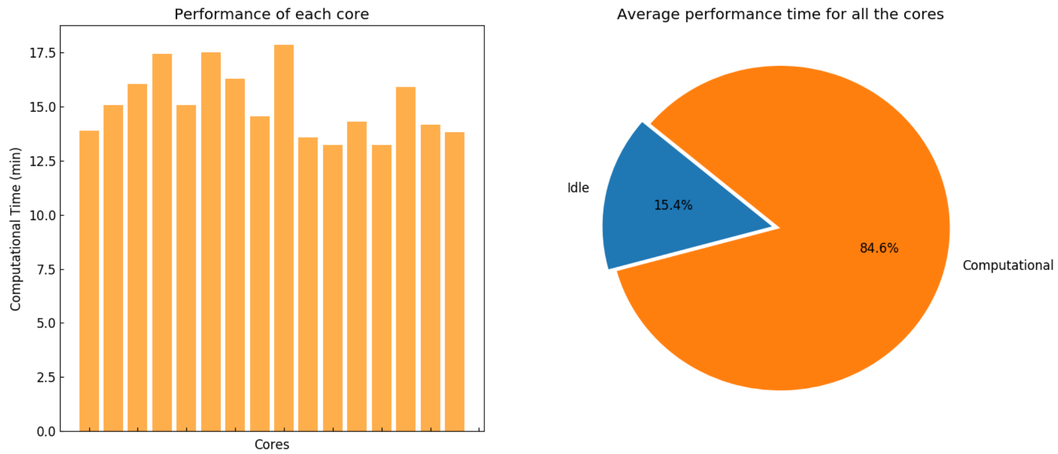


Figure 3: (a) Computational time versus number of lines, (b) Memory used per core vs the number of lines

From figure 3a it is clear that as the number of lines increases the computational time of the code increases, what is interesting is that the run on a single or on a multiple machine with the same number of cores had the same amount of computational time. The reason behind this behavior is because the MPI code is implemented in a way where cores do not rely on each other, and the communication time between the different machines was reduced to almost null, enabling both machines to behave in a similar manner. A better example to explain this behavior is shown in figure 4, where the performance of a 16 core on a single machine is displayed versus the performance of a multiple machine for a 1 million line. The results show that on average the cores on single or multiple machine operate at the same speed. However there are some cores on the multiple machine that took double the time, this can be attributed to the fact that the cashe on that machine were being used by another user. But on general the mpi code on single and multiple machine operate at the same time. The downside of this approach is that all the cores will take the same amount of memory, even though they will not use all of it as is shown in figure 3b. A better way was to store only the memory of that will be needed but that will be a hassle to code

(a) *Signle Machine*



(b) *Multiple Machines*

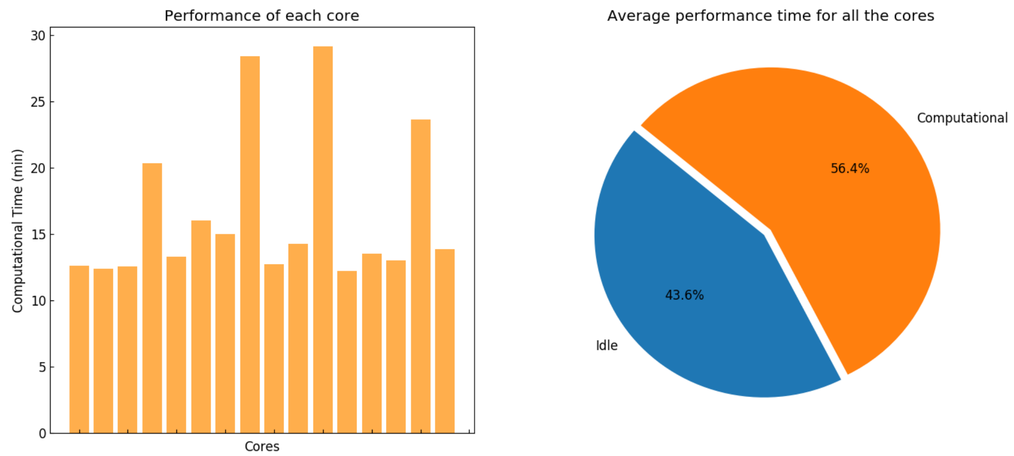


Figure 4: Computational time for each core and how much on average did the cores work or stay idel for number of lines equal 1,000,000 on a (a) single machine, (b) multiple machines

Figures 5a and 5b show the performance of the code versus the number of cores for the case with lines equal to 1,000,000. The figure clearly shows that as the number of cores increases the run will take less time starting with around 3.75 hrs for 1 core, and ending with around 8 minutes for 64 cores.

(a) Memory Usage

(b) Final Answer

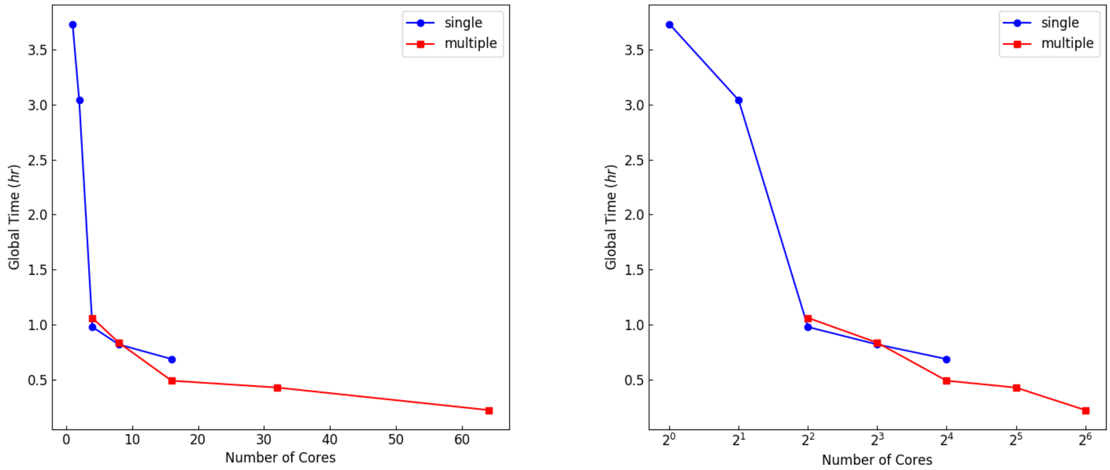


Figure 5: (a) Performance of the code versus the number of cores for the case with lines equal to 1,000,000, (b) the same but log_2 for the x-axis

5 References:

1.$https://support.beocat.ksu.edu/BeocatDocs/index.php/Main_Page$
2.$https://support.beocat.ksu.edu/BeocatDocs/index.php/Compute_Nodes$

6 Appendice:

Listing 1: 'Find_keys_mpi.c'

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include <mpi.h>
#include <sys/resource.h>

# define version 1

int maxwords = 50000;
int maxlines;
int nwords;
int nlines;
int err, *count, nthreads = 1;
double tstart, ttotal;
FILE *fd;
char **word, **line;
char *key_array;
long key_array_size;
int filenumber;
char newword[15], hostname[256],filename[256];

/* myclock: (Calculates the time)
--------------------------------------------*/

double myclock() {
    static time_t t_start = 0;  // Save and subtract off each time

    struct timespec ts;
    clock_gettime(CLOCK_REALTIME, &ts);
    if( t_start == 0 ) t_start = ts.tv_sec;

    return (double) (ts.tv_sec - t_start) + ts.tv_nsec * 1.0e-9;
}

/* init_list: (Initaite word list and lines)
--------------------------------------------*/
void init_list(void *rank)
{
    // Malloc space for the word list and lines
    int i;
    int myID =  *((int*) rank);
    word = (char **) malloc( maxwords * sizeof( char * ) );
    count = (int *) malloc( maxwords * sizeof( int ) );
    for( i = 0; i < maxwords; i++ ) {
        word[i] = malloc( 10 );
        count[i] = 0;
    }

    line = (char **) malloc( maxlines * sizeof( char * ) );
    for( i = 0; i < maxlines; i++ ) {
        line[i] = malloc( 2001 );
```

```
    }

    key_array_size = 1000*maxlines;
    key_array = malloc(sizeof(char)*key_array_size);
    key_array[0] = '\0';
}


/* read_dict_words: (Read Dictianary words)
--------------------------------------------*/
void read_dict_words()
{
// Read in the dictionary words
    fd = fopen("625/keywords.txt", "r" );
    nwords = -1;
    do {
        err = fscanf( fd, "%[^\n]\n", word[++nwords] );
    } while( err != EOF && nwords < maxwords );
    fclose( fd );

    //printf( "Read in %d words\n", nwords);
}


/* read_lines: (Read wiki lines)
--------------------------------------------*/
void read_lines()
{
// Read in the lines from the data file
    double nchars = 0;
    fd = fopen( "625/wiki_dump.txt", "r" );
    nlines = -1;
    do {
        err = fscanf( fd, "%[^\n]\n", line[++nlines] );
        if( line[nlines] != NULL ) nchars += (double) strlen( line[nlines] );
    } while( err != EOF && nlines < maxlines);
    fclose( fd );
}

/* reset_file: (create an output file)
--------------------------------------------*/
void *reset_file()
{
    snprintf(filename,250,"wiki-mpi-%d.out", filenumber);
    fd = fopen(filename, "w" );
    fclose( fd );
}

/* count_words: (count the words)
--------------------------------------------*/
void *count_words(void *rank)
{
    int i,k;
    int myID =  *((int*) rank);
    int startPos = ((long) myID) * (nwords / nthreads);
```

```
    int endPos = startPos + (nwords / nthreads);

    for( i = startPos; i < endPos; i++ ) {
        for( k = 0; k < nlines; k++ ) {
            if( strstr( line[k], word[i] ) != NULL )
            {
                if (count[i] == 0)
                {
                    snprintf(newword,8,"%s:",word[i]);
                    strcat(key_array,newword);
                }
                count[i]++;
                snprintf(newword,8,"%d,",k);
                strcat(key_array,newword);
            }
        }
        //if (i % 1000 == 0) printf("Loop %d of %d done\n", i, nwords);
        if (count[i] != 0)
        {
            snprintf(newword,5,"\n");
            strcat(key_array,newword);
        }
    }
}


/* dump_words: (write the output on file)
--------------------------------------------*/
void *dump_words()
{
    fd = fopen(filename, "a" );
    int results =fputs(key_array,fd);
    fclose( fd );
}


/*--------------------------------------------------
                        Main
--------------------------------------------------*/

int main(int argc, char* argv[])
{

    int i, rc;
    int numtasks, rank;
    double tstart_init, tend_int, tstart_count, tend_count, tend_reduce;
    struct rusage ru;

    MPI_Status Status;
    maxlines = 100000; // Default Value
    filenumber = rand() %100000;
    if (argc >= 2){
        maxlines = atol(argv[1]);
        filenumber = atol(argv[2]); // Name of the output file
    }
```

```
    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,&numtasks); // Number of cores
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);     // rank of each core
    nthreads = numtasks;

    gethostname(hostname,255);
    // The last core will reset the output file
    if(rank == nthreads-1) reset_file();

    tstart = myclock(); // Global Clock
    // Initialization
    tstart_init = MPI_Wtime();  // Private Clock for each core
    init_list(&rank);
    read_dict_words();
    read_lines();
    tend_int    = MPI_Wtime();

    getrusage(RUSAGE_SELF, &ru);
    long MEMORY_USAGE = ru.ru_maxrss;   // Memory usage in Kb
    // Counting
    tstart_count = MPI_Wtime();
    count_words(&rank);
    tend_count = MPI_Wtime();

    // Outputing on File
    tend_reduce = MPI_Wtime();
    for(i = 0; i < nthreads;++i)
    {
        if(rank == i)   dump_words();
        MPI_Barrier(MPI_COMM_WORLD);
    }
    printf("initialization time %lf, counting time %lf, writing time %lf,
    size %d, rank %d, hostaname %s, memory %ld Kb\n", tend_int - tstart_init,
     tend_count - tstart_count, tend_reduce - tend_count, numtasks,
     rank, hostname, MEMORY_USAGE);

    fflush(stdout);

    if ( rank == 0 ) {
    ttotal = myclock() - tstart;
    printf("version %d, cores %d, total time %lf seconds, words %d,
     lines %d\n", version, nthreads, ttotal, nwords, maxlines);
    }
    MPI_Finalize();
    return 0;
}
```

Listing 2: Bash script example for multiple

```
#!/bin/bash
#$ -l mem=1G
#$ -l h_rt=24:00:00
#$ -l killable
#$ -cwd
#$ -q \*@@elves
#$ -pe mpi-2 64

for i in 1000000 800000 600000 400000 200000
do
mpirun -np 64 /homes/mcheikh/CIS_625/hw3/MPI_V2.out $i 264
hostname
echo -e "---Done---\n"
done
##/homes/mcheikh/CIS_625/hw2/a.out
```

Listing 3: Bash script example for single

```
#!/bin/bash
#$ -l mem=1G
#$ -l h_rt=24:00:00
#$ -l killable
#$ -cwd
#$ -q \*@@elves
#$ -pe signle 16

for i in 1000000 800000 600000 400000 200000
do
mpirun -np 16 /homes/mcheikh/CIS_625/hw3/MPI_V2.out $i 168
hostname
echo -e "---Done---\n"
done
##/homes/mcheikh/CIS_625/hw2/a.out
```