Getting fancier with 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 implement three sophisticated distributed parallel execution models, and analyze their performance on a machine like Beocat.

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 three different communication paradigms implemented for distributed programming. All of the codes were compiled using the Open MPI C wrapper compiler (mpicc) in Beocat, with a second level optimization (-O2).

3.1 Star/Centralized

The first implementation is the *Star/Centralized* approach, this approach follows the master, slave relation. In our case the master core (which is rank 0) will read all the keywords, and distribute them one by one over the rest of the cores. As for the slave cores, they will take the keyword from the master core, and look it up in their own respective Wikipedia lines. After the slave cores finish looking up the keyword they will send their output to the master core and wait for the next keyword. At the end of the run, the master core will compile and sort all the outputs it received and print them out. The figure below gives a clearer picture of how the program is working

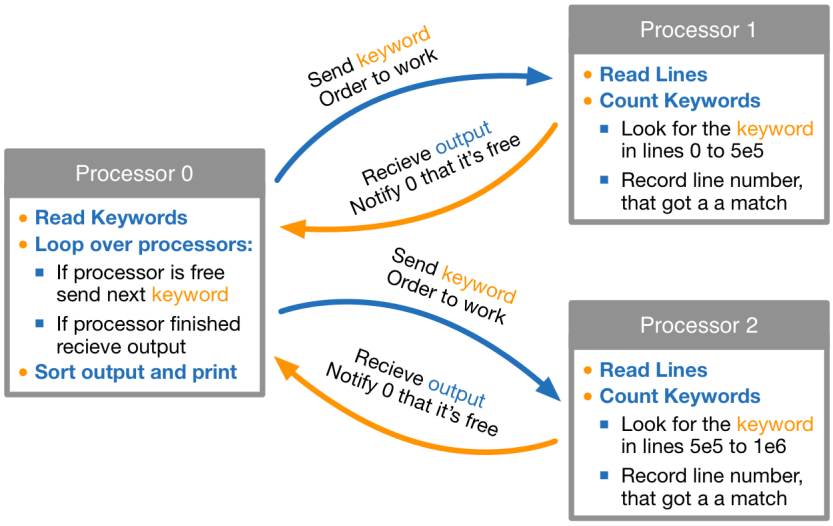


Figure 1: Diagram depicting the *Star/centralized* approach

3.2 Ring

The second distributed programming implementation is the *Ring* approach, in this approach the task starts from process 0, cycles through all the cores and returns back to process 0 to be printed. In our case process 0, will read all the keywords, and send the keywords one by one to the next processor, which is processor 1. The next processor will receive the new keyword, and look it up in its own respective Wikipedia lines. Then the processor will send the keyword, along with the output it got to the next processor and so on. At the end when the keyword returns back to process 0, it will have with it the output of all the processors, and since it went in numerical order then no sorting is required. The output will be printed out directly. The figure below gives a clearer picture of how the program is working

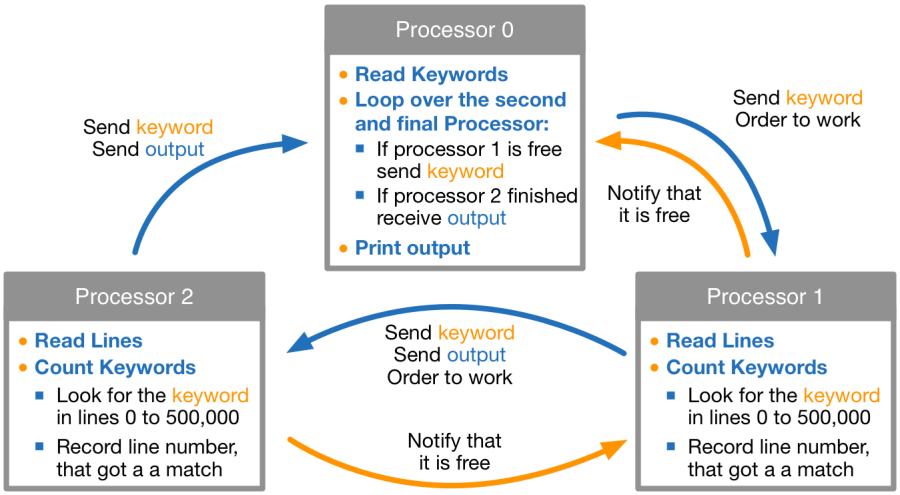


Figure 2: Diagram depicting the *Ring* approach

3.3 Work Queue

The final distributed programming implementation is the *Work queue* approach, this approach also follows the master/slave relation but in a different manner than the *Star/centralized* approach. In this approach processor 0 (which is the master process) will batch up the keywords into groups of hundred, and send the batches to any free or idle processors. The slave processors will look up the keywords they received in the all of Wikipedia lines, and record the number and location of matches. When the look up process is over, the processor will notify processor 0 to send the next batch of keywords. At the end of the program all the slave processors will send their output to processor 0 to sort them out and print them.

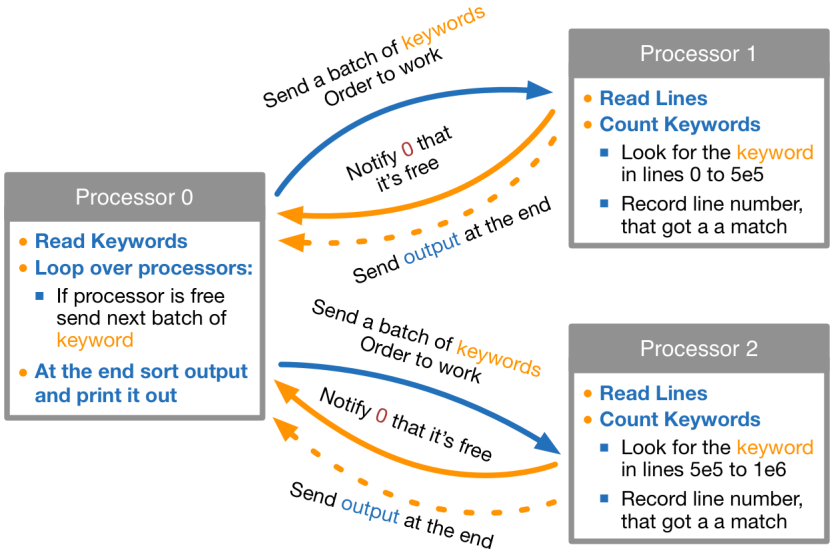
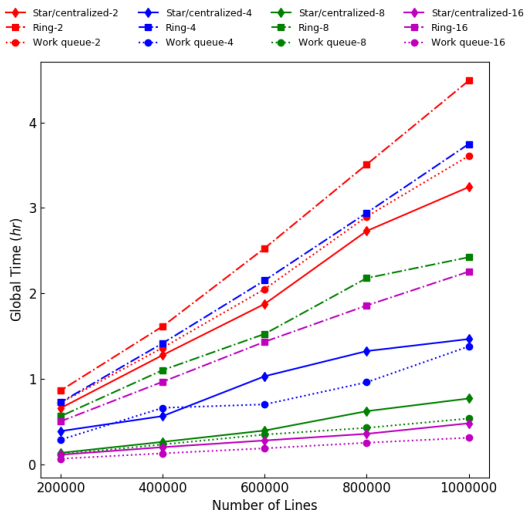


Figure 3: Diagram depicting the *Work queue* approach

4 Results

The three approaches discussed in the previous section were tested for different number of lines starting from $200,000$ and going up to $1,000,000$ with an increment of $200,000$, and with different number of cores (2, 4, 8, and 16). The results of the run are shown in figures 4.

(a) Computational Time



(b) Memory

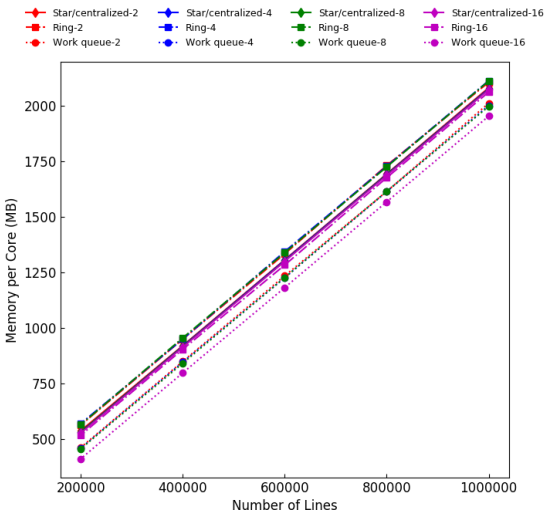


Figure 4: (a) Computational time versus number of lines, (b) Memory used per core vs the number of lines

From figure 4a it is clear that the fastest of threes approaches is the *Work queue* approach, followed by the *Star/centralized*, and the slowest is the *Ring* approach. The reason why the *Work queue* approach is the fastest is because it has a lower communication time than the *Star/centralized* approach, and the processes don't depend on each other like the *Ring* approach. As for the *Ring* approach, it is the slowest because all the processes have to wait until the previous processor has finished working on the keyword so they can start working on it. Thus by waiting on the process to finish, the task becomes increasingly serial.

Figures 5 shows the performance of the three approaches 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 computational time for the *Star/centralized* and the *Work queue* decrease, as for the *Ring* approach it initially decreases drastically then from 8 cores to 16 cores to decreases slower. This is because as the number of cores increases in the *Ring* approach the computational time decreases, but the total communication time between all the cores increases (Since the communication time will stay the same between two cores, but we have an increase in the core number, then the total communication time between all the cores increases). Since the *Ring* approach is highly dependent on the on the communication, then an increase in the total communication time will cause it to get slower. Moreover after a certain number of cores the comm to comp ratio per core for the *Ring* will approach a value greater than one, and so increasing the number of cores will not benefit the solver, it may slow it down. In our case the breaking point was eight cores.

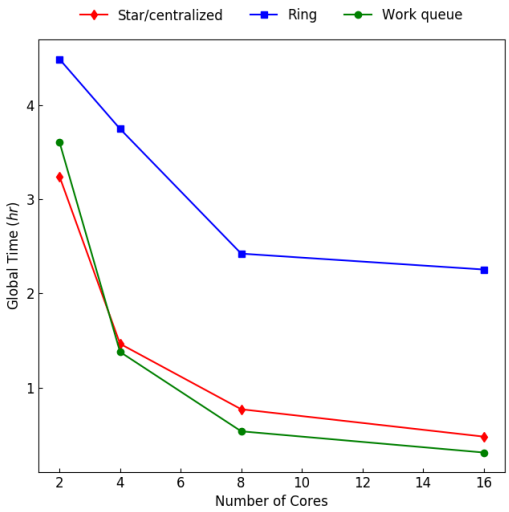


Figure 5: Computational time versus number of cores for number of lines equal to 1,000,000.

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 Appendix:

Listing 1: 'Find_keys_mpi_Star.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, **all_key_array;
char *key_array;
long key_array_size, key_array_count, key_array_limit,
    new_key_array_size;
int filenumber;
char newword[15], hostname[256],filename[256], sent_word[20],
    word_to_send[20];
int *Free, *word_number , w_number;

/* 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);
  count = 0;
  word = (char **) malloc( maxwords * sizeof( char * ) );
  for( i = 0; i < maxwords; i++ ) {
    word[i] = malloc( 10 );
  }

  line = (char **) malloc( maxlines * sizeof( char * ) );
```

```
51    for( i = 0; i < maxlines; i++ ) {
52      line[i] = malloc( 2001 );
53    }
54
55      key_array_count = 0;       // counter
56      key_array_limit =  99000000; // limit to increase array size: 99
          million
57      key_array_size  = 100000000; // size of key_array: 100 million ~ 95
           MB
58      new_key_array_size = key_array_size; // new array size (initail
          value is key_array_size)
59    key_array = malloc(sizeof(char)*key_array_size);
60    key_array[0] = '\0';
61    sent_word[0] = '\0';
62    word_to_send[0] = '\0';
63
64    // This is an array that tells rank 0 which processor is free
65    Free = (int *) malloc( nthreads * sizeof( int ) ); // 0 free, 1 busy
66    word_number = (int *) malloc( nthreads * sizeof( int ) );
67    for (i = 0; i < nthreads; i++)
68    {
69      Free[i] = 0; //Initial values (all are free)
70      word_number[i] = 0;
71    }
72
73    if (myID == 0)
74    {
75      all_key_array = (char **) malloc( (nthreads-1) * sizeof( char * ) )
          ;
76      for( i = 0; i < nthreads-1; i++ ) {
77        all_key_array[i] = malloc( key_array_size );
78        all_key_array[i][0] = '\0';
79      }
80    }
81  }
82
83
84  /* read_dict_words: (Read Dictianary words)
85  ----------------------------------------------*/
86  void read_dict_words()
87  {
88  // Read in the dictionary words
89    fd = fopen("625/keywords.txt", "r" );
90    nwords = -1;
91    do {
92      err = fscanf( fd, "%[^\n]\n", word[++nwords] );
93    } while( err != EOF && nwords < maxwords );
94    fclose( fd );
95
96    //printf( "Read in %d words\n", nwords);
97  }
98
99
100 /* read_lines: (Read wiki lines)
101 ----------------------------------------------*/
```

```
102 void read_lines()
103 {
104 // Read in the lines from the data file
105   double nchars = 0;
106   fd = fopen( "625/wiki_dump.txt", "r" );
107   nlines = -1;
108   do {
109     err = fscanf( fd, "%[^\n]\n", line[++nlines] );
110     if( line[nlines] != NULL ) nchars += (double) strlen( line[nlines]
          );
111   } while( err != EOF && nlines < maxlines);
112   fclose( fd );
113
114   //printf( "Read in %d lines averaging %.0lf chars/line\n", nlines,
        nchars / nlines);
115 }
116
117 /* reset_file: (create an output file)
118 ------------------------------------------*/
119 void *reset_file()
120 {
121   snprintf(filename,250,"wiki-mpi-%d.out", filenumber);
122   fd = fopen(filename, "w" );
123   fclose( fd );
124 }
125
126 /* count_words: (count the words)
127 ------------------------------------------*/
128 void *count_words(void *rank)
129 {
130   int i,k;
131   int myID =  *((int*) rank);
132   int startPos = ((long) myID-1) * (nlines / (nthreads-1));
133   int endPos = startPos + (nlines / (nthreads-1));
134   count = 0;
135   for( k = startPos; k < endPos; k++ )
136     {
137       if (count < 100)
138       {
139         if( strstr( line[k], sent_word ) != NULL )
140         {
141           if (count == 0)
142           {
143             //snprintf(newword,10,"%s:",sent_word);
144             snprintf(newword,10,"X%d:",w_number);
145             strcat(key_array,newword);
146             key_array_count += strlen(newword);
147           }
148
149           count++;
150           snprintf(newword,10,"%d,",k);
151           strcat(key_array,newword);
152           key_array_count += strlen(newword);
153         }
154       }
```

```
155
156       }
157
158         // Checks if the size of the key_array has reached the limit
159       if (key_array_count > key_array_limit)
160       {
161         new_key_array_size += key_array_size; // incrase size
162         char* myrealloced_array = realloc(key_array, new_key_array_size *
              sizeof(char));
163         if (myrealloced_array) key_array = myrealloced_array;
164         key_array_count = 0; // reset counter
165       }
166   }
167
168
169   /* dump_words: (write the output on file)
170   ----------------------------------------------*/
171   void *dump_words()
172   {
173     fd = fopen(filename, "a" );
174     int results =fputs(key_array,fd);
175     fclose( fd );
176   }
177
178
179   void *remove_elements(char *array, int array_length, int index)
180   {
181      int i;
182      for(i = 0; i < array_length - index; i++)
183      {
184       array[i] = array[i+index];
185      }
186
187      for(i= array_length - index; i<array_length; i++)
188     {
189       array[i] = '\0';
190     }
191   }
192
193   /*-------------------------------------------------
194               Main
195   -------------------------------------------------*/
196
197   int main(int argc, char* argv[])
198   {
199
200     int i, rc;
201     int numtasks, rank;
202     double tstart_init, tend_int, tstart_count, tend_count, tend_reduce;
203       struct rusage ru;
204
205     MPI_Status Status;
206       MPI_Request Request;
207     maxlines = 100000; // Default Value
208     filenumber = rand() %100000;
```

```
209   if (argc >= 2){
210     maxlines = atol(argv[1]);
211     filenumber = atol(argv[2]);
212   }
213
214   rc = MPI_Init(&argc,&argv);
215   if (rc != MPI_SUCCESS){
216     printf ("Error starting MPI program. Terminating.\n");
217     MPI_Abort(MPI_COMM_WORLD, rc);
218   }
219
220     MPI_Comm_size(MPI_COMM_WORLD,&numtasks); // Number of cores
221     MPI_Comm_rank(MPI_COMM_WORLD,&rank);    // rank of each core
222   nthreads = numtasks;
223
224   gethostname(hostname,255);
225   if(rank == nthreads-1) reset_file(); // The last core will reset the
        output file
226   MPI_Bcast(filename, 256, MPI_CHAR, nthreads-1, MPI_COMM_WORLD); //
        send the name of the output file to all cores
227   tstart = myclock(); // Global Clock
228
229   // Initialization
230   init_list(&rank);
231
232   if (rank ==0) read_dict_words();
233   else           read_lines();
234
235   int flag = 0;
236   int flag2 = 0;
237   int DONE = 0;
238   MPI_Barrier(MPI_COMM_WORLD);
239   w_number = 0;
240   int receiver = 0;
241
242
243   while (DONE < nthreads -1)
244   {
245     // Sending keywords from rank 0 to all the other:
246     //-----------------------------------------------
247     if (rank == 0)
248     {
249       for (receiver = 1; receiver < nthreads; receiver++)
250       {
251         if (word_number[receiver] < maxwords)
252         {
253           DONE = 0;
254           if (Free[receiver] == 0)
255           {
256             strcat(word_to_send, word[word_number[receiver]]);
257             word_number[receiver] += 1;
258             Free[receiver] = 1; // Means that the reciever is now busy
259             MPI_Isend(&Free[receiver], 1, MPI_INT, receiver, 1234,
                  MPI_COMM_WORLD, &Request);
```

```
260             MPI_Isend(word_to_send,   20, MPI_CHAR, receiver, 5678,
                    MPI_COMM_WORLD, &Request);
261             memset(&word_to_send[0], 0, sizeof(word_to_send));
262           }
263           else
264           { // Check if other ranks sent their output:
265             //---------------------------------------
266             MPI_Iprobe(receiver, 4321, MPI_COMM_WORLD, &flag, &Status )
                    ;
267             if (flag)
268             {
269               MPI_Irecv(&Free[receiver], 1, MPI_INT, receiver, 4321,
                      MPI_COMM_WORLD, &Request);
270               while(!flag2)
271                   {
272                     MPI_Iprobe(receiver, 7777, MPI_COMM_WORLD, &flag2,
                        &Status );
273                   }
274                   MPI_Irecv(key_array, key_array_size, MPI_CHAR,
                      receiver, 7777, MPI_COMM_WORLD, &Request);
275
276                   if (strlen(key_array) > 0)
277                   {
278                     strcat(key_array,"\n");
279                     strcat(all_key_array[receiver-1],key_array);
280                   }
281                   memset(&key_array[0], 0, sizeof(key_array)*strlen(
                      key_array) );
282               flag2 = 0;
283             }
284             flag = 0;
285           }
286         }
287         else
288         {
289           DONE+=1;
290         }
291       }
292     }
293
294     // Receiving  keywords from rank 0:
295     //-------------------------------------------------
296     else
297     {
298       MPI_Iprobe(0, 1234, MPI_COMM_WORLD, &flag, &Status );
299       if (flag)
300       {
301         MPI_Irecv(&Free[rank], 1, MPI_INT, 0, 1234, MPI_COMM_WORLD, &
                Request);
302         if (Free[rank] == 1)
303         {
304           while(!flag2)
305               {
306                 MPI_Iprobe( 0, 5678, MPI_COMM_WORLD, &flag2, &Status );
307               }
```

```
308             MPI_Irecv(&sent_word, 20, MPI_CHAR, 0, 5678, MPI_COMM_WORLD,
                    &Request);
309             count_words(&rank); // Counting
310             flag2 = 0;
311             flag = 0;
312             Free[rank] = 0; // Means that the reciever is now free
313             MPI_Isend(&Free[rank], 1, MPI_INT,0,4321,MPI_COMM_WORLD, &
                    Request);  // Tell rank 0 you r free
314             MPI_Isend(key_array, strlen(key_array), MPI_CHAR, 0, 7777,
                    MPI_COMM_WORLD, &Request);
315              memset(&key_array[0], 0, sizeof(key_array)*strlen(
                     key_array) );
316              w_number+=1;
317           }
318         }
319         MPI_Iprobe(0, 9999, MPI_COMM_WORLD, &flag, &Status );
320         if (flag)
321         {
322           MPI_Irecv(&DONE, 1, MPI_INT, 0, 9999, MPI_COMM_WORLD, &Request)
                ;
323         }
324       }
325     }
326
327     // Exiting the counting Processs:
328     //-----------------------------------------------
329
330     if(rank==0)
331     {
332       for (receiver = 1; receiver < nthreads; receiver++)
333       {
334         MPI_Isend(&DONE, 1, MPI_INT, receiver,9999, MPI_COMM_WORLD, &
              Request);
335       }
336     }
337
338     printf("Done rank %d\n",rank);
339
340
341     // Sorting the results and Printing the output:
342     //-----------------------------------------------
343
344     if(rank==0)
345     {
346       char integer[32];
347       char output_word[32];
348       int results;
349       int number_of_times = 0;
350       memset(&key_array[0], 0, sizeof(key_array)*strlen(key_array) );
351
352       fd = fopen(filename, "a" );
353       char *look_for_word;
354
355       for (i = 0; i < maxwords ; i++)
356       {
```

```
357         sprintf(integer, "X%d:",i);
358         strcat(output_word, word[i]);
359         strcat(output_word, ":");
360         for (receiver = 1; receiver < nthreads; receiver++)
361         {
362
363           look_for_word = strstr(all_key_array[receiver-1], integer);
                    // search for string
364           if (look_for_word != NULL)                    // if successful
365           {
366
367             sscanf(look_for_word, "%[^\n]", key_array);
368
369             if(number_of_times == 0) results = fputs(output_word, fd);
370             remove_elements(key_array,strlen(key_array),strlen(integer));
371             results = fputs(key_array,fd);
372             number_of_times += 1;
373           }
374           look_for_word = NULL;
375           memset(&key_array[0], 0, sizeof(key_array)*strlen(key_array) );
376         }
377         if (number_of_times > 0)
378         {
379           results = fputs("\n",fd);
380         }
381         number_of_times = 0;
382             memset(&output_word[0], 0, sizeof(char)*strlen(output_word) )
                ;
383             memset(&integer[0], 0, sizeof(char)*strlen(integer) );
384       }
385       fclose( fd );
386     }
387     getrusage(RUSAGE_SELF, &ru);
388       long MEMORY_USAGE = ru.ru_maxrss;   // Memory usage in Kb
389     printf(" rank %d, hostaname %s, size %d, memory %ld Kb\n",
390                 rank, hostname, nthreads,  MEMORY_USAGE);
391     fflush(stdout);
392
393     if ( rank == 0 ) {
394     ttotal = myclock() - tstart;
395     printf("version %d, cores %d, total time %lf seconds, words %d, lines
          %d\n",
396         version, nthreads, ttotal, nwords, maxlines);
397     }
398
399     MPI_Finalize();
400     return 0;
401   }
```

Listing 2: 'Find_keys_mpi_Ring.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, key_array_count, key_array_limit,
    new_key_array_size;
int filenumber;
char newword[15], hostname[256],filename[256], sent_word[20];
int  word_number;

/* 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);
  count = 0;
  word = (char **) malloc( maxwords * sizeof( char * ) );
  for( i = 0; i < maxwords; i++ ) {
    word[i] = malloc( 10 );
  }

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

```
54
55      key_array_count = 0;        // counter
56      key_array_limit =  99000000; // limit to increase array size: 99
            million
57      key_array_size  = 100000000; // size of key_array: 100 million ~ 95
            MB
58      new_key_array_size = key_array_size; // new array size (initail
            value is key_array_size)
59    key_array = malloc(sizeof(char)*key_array_size);
60    key_array[0] = '\0';
61    sent_word[0] = '\0';
62  }
63
64
65  /* read_dict_words: (Read Dictianary words)
66  ------------------------------------------*/
67  void read_dict_words()
68  {
69  // Read in the dictionary words
70    fd = fopen("625/keywords.txt", "r" );
71    nwords = -1;
72    do {
73      err = fscanf( fd, "%[^\n]\n", word[++nwords] );
74    } while( err != EOF && nwords < maxwords );
75    fclose( fd );
76
77    //printf( "Read in %d words\n", nwords);
78  }
79
80
81  /* read_lines: (Read wiki lines)
82  ------------------------------------------*/
83  void read_lines()
84  {
85  // Read in the lines from the data file
86    double nchars = 0;
87    fd = fopen( "625/wiki_dump.txt", "r" );
88    nlines = -1;
89    do {
90      err = fscanf( fd, "%[^\n]\n", line[++nlines] );
91      if( line[nlines] != NULL ) nchars += (double) strlen( line[nlines]
            );
92    } while( err != EOF && nlines < maxlines);
93    fclose( fd );
94
95    //printf( "Read in %d lines averaging %.0lf chars/line\n", nlines,
          nchars / nlines);
96  }
97
98  /* reset_file: (create an output file)
99  ------------------------------------------*/
100 void *reset_file()
101 {
102   snprintf(filename,250,"wiki-mpi-%d.out", filenumber);
103   fd = fopen(filename, "w" );
```

```
104   fclose( fd );
105 }
106
107 /* count_words: (count the words)
108 ------------------------------------------*/
109 void *count_words(void *rank)
110 {
111   int i,k;
112   int myID =  *((int*) rank);
113   int startPos = ((long) myID-1) * (nlines / (nthreads-1));
114   int endPos = startPos + (nlines / (nthreads-1));
115   key_array_count = 0;
116   for( k = startPos; k < endPos; k++ )
117     {
118       if (count < 100)
119       {
120         if( strstr( line[k], sent_word ) != NULL )
121         {
122           if (count == 0)
123           {
124             snprintf(newword,10,"%s:",sent_word);
125             strcat(key_array,newword);
126             key_array_count += strlen(newword);
127           }
128
129           count++;
130           snprintf(newword,10,"%d,",k);
131           strcat(key_array,newword);
132           key_array_count += strlen(newword);
133         }
134       }
135
136     }
137
138       // Checks if the size of the key_array has reached the limit
139     if (key_array_count > key_array_limit)
140     {
141       new_key_array_size += key_array_size; // incrase size
142       char* myrealloced_array = realloc(key_array, new_key_array_size *
              sizeof(char));
143       if (myrealloced_array) key_array = myrealloced_array;
144       key_array_count = 0; // reset counter
145     }
146 }
147
148
149 /* dump_words: (write the output on file)
150 ------------------------------------------*/
151 void *dump_words()
152 {
153   fd = fopen(filename, "a" );
154   int results =fputs(key_array,fd);
155   results =fputs("\n",fd);
156   fclose( fd );
157 }
```

```
158
159
160 void *remove_elements(char *array, int array_length, int index)
161 {
162    int i;
163    for(i = 0; i < array_length - index; i++)
164    {
165     array[i] = array[i+index];
166    }
167 }
168
169 /*-------------------------------------------------
170             Main
171 -------------------------------------------------*/
172
173 int main(int argc, char* argv[])
174 {
175
176   int i, rc;
177   int numtasks, rank;
178   double tstart_init, tend_int, tstart_count, tend_count, tend_reduce;
179     struct rusage ru;
180
181   MPI_Status Status;
182     MPI_Request Request;
183   maxlines = 100000; // Default Value
184   filenumber = rand() %100000;
185   if (argc >= 2){
186     maxlines = atol(argv[1]);
187     filenumber = atol(argv[2]);
188   }
189
190   rc = MPI_Init(&argc,&argv);
191   if (rc != MPI_SUCCESS){
192     printf ("Error starting MPI program. Terminating.\n");
193     MPI_Abort(MPI_COMM_WORLD, rc);
194   }
195
196     MPI_Comm_size(MPI_COMM_WORLD,&numtasks); // Number of cores
197     MPI_Comm_rank(MPI_COMM_WORLD,&rank);   // rank of each core
198   nthreads = numtasks;
199
200   gethostname(hostname,255);
201   if(rank == nthreads-1) reset_file(); // The last core will reset the
        output file
202   MPI_Bcast(filename, 256, MPI_CHAR, nthreads-1, MPI_COMM_WORLD); //
        send the name of the output file to all cores
203   tstart = myclock(); // Global Clock
204
205   // Initialization
206   init_list(&rank);
207   if (rank ==0) read_dict_words();
208   else          read_lines();
209   int flag = 0;
210   int flag2 = 0;
```

```
211   int DONE = 0;
212   MPI_Barrier(MPI_COMM_WORLD);
213
214   word_number = 0;
215   int words_written =0;
216
217   int receiver,sender;
218
219   if(rank==nthreads-1) receiver = 0; // reciever is the rank that will
          recieve the message
220   else    receiver = rank + 1;
221
222   if(rank==0) sender = nthreads-1; // sender is the rank that will send
          the message
223   else    sender = rank - 1;
224
225   int Waiting = 0;
226   // Task to be done by rank 0 before starting the loop:
227   if (rank==0)
228   {
229     strcat(sent_word, word[word_number]);
230     word_number +=1;
231     count = 0;
232     //Free[receiver]=1;
233     //MPI_Isend(&Free[receiver], 1, MPI_INT, receiver, 1234,
            MPI_COMM_WORLD, &Request);
234     MPI_Isend(&count, 1, MPI_INT, receiver, 2233, MPI_COMM_WORLD, &
            Request);
235     MPI_Isend(sent_word,   20, MPI_CHAR, receiver, 5678, MPI_COMM_WORLD
            , &Request);
236     MPI_Isend(key_array, strlen(key_array), MPI_CHAR, receiver, 7777,
            MPI_COMM_WORLD, &Request);
237   }
238
239   // Entering the while loop:
240   while(DONE != 1)
241   {
242     // Check fo int count
243     MPI_Iprobe(sender, 2233, MPI_COMM_WORLD, &flag2, &Status );
244     if(flag2)
245     {
246       MPI_Irecv(&count, 1, MPI_INT, sender, 2233, MPI_COMM_WORLD, &
              Request);
247       flag2 = 0;
248       // Recieve char sent_word
249       while(!flag2)
250       {
251       MPI_Iprobe( sender, 5678, MPI_COMM_WORLD, &flag2, &Status );
252       }
253       MPI_Irecv(&sent_word, 20, MPI_CHAR, sender, 5678, MPI_COMM_WORLD,
              &Request);
254       flag2 = 0;
255
256       // Recieve char key_array
257       while(!flag2)
```

```
258         {
259         MPI_Iprobe( sender, 7777, MPI_COMM_WORLD, &flag2, &Status );
260         }
261         MPI_Irecv(key_array, key_array_size, MPI_CHAR, sender, 7777,
                MPI_COMM_WORLD, &Request);
262         flag2 = 0;
263
264         // Tasks to do
265         if(rank == 0)
266         {
267           if(strlen(key_array)>0) dump_words();
268           // Reset
269           Waiting = 0;
270           words_written +=1;
271           if(words_written==maxwords) DONE = 1;
272         }
273         else
274         {
275           count_words(&rank); // CHECK THIS WORK
276         }
277
278         // Send again
279         if (rank != 0)
280         {
281           MPI_Isend(&count, 1, MPI_INT, receiver, 2233, MPI_COMM_WORLD, &
                Request);
282           MPI_Isend(sent_word,   20, MPI_CHAR, receiver, 5678,
                MPI_COMM_WORLD, &Request);
283           MPI_Isend(key_array, strlen(key_array), MPI_CHAR, receiver,
                7777, MPI_COMM_WORLD, &Request);
284           memset(&key_array[0], 0, sizeof(key_array[0])*strlen(key_array)
                );
285           key_array[0] = '\0';
286         }
287
288       }
289       MPI_Iprobe(0, 9999, MPI_COMM_WORLD, &flag, &Status );
290       if (flag)
291       {
292         MPI_Irecv(&DONE, 1, MPI_INT, 0, 9999, MPI_COMM_WORLD, &Request);

293       }
294
295       if (rank ==0 && Waiting <1 && word_number<maxwords)
296       {
297         // Reset
298         memset(&key_array[0], 0, sizeof(key_array[0])*strlen(key_array));
299         memset(&sent_word[0], 0, sizeof(sent_word[0])*strlen(sent_word));

300         count = 0;
301         // New word
302
303         strcat(sent_word, word[word_number]);
304         MPI_Isend(&count, 1, MPI_INT, receiver, 2233, MPI_COMM_WORLD, &
              Request);
```

```
305         MPI_Isend(sent_word,   20, MPI_CHAR, receiver, 5678,
                MPI_COMM_WORLD, &Request);
306         MPI_Isend(key_array, strlen(key_array), MPI_CHAR, receiver, 7777,
                 MPI_COMM_WORLD, &Request);
307         memset(&key_array[0], 0, sizeof(key_array[0])*strlen(key_array));

308         key_array[0] = '\0';
309
310         word_number +=1;
311         Waiting +=1;
312       }
313     }
314
315     // Exiting the counting Processs:
316     //-----------------------------------------------
317     if(rank==0)
318     {
319       for (receiver = 1; receiver < nthreads; receiver++)
320       {
321         MPI_Isend(&DONE, 1, MPI_INT, receiver,9999, MPI_COMM_WORLD, &
                Request);
322       }
323     }
324     printf("Done rank %d\n",rank);
325
326     // Print results:
327     //-----------------------------------------------
328
329     getrusage(RUSAGE_SELF, &ru);
330       long MEMORY_USAGE = ru.ru_maxrss;   // Memory usage in Kb
331     printf(" rank %d, hostaname %s, size %d, memory %ld Kb\n",
332                 rank, hostname, nthreads,  MEMORY_USAGE);
333     fflush(stdout);
334
335     if ( rank == 0 ) {
336     ttotal = myclock() - tstart;
337     printf("version %d, cores %d, total time %lf seconds, words %d, lines
            %d\n",
338         version, nthreads, ttotal, nwords, maxlines);
339     }
340
341     MPI_Finalize();
342     return 0;
343   }
```

Listing 3: 'Find_keys_mpi_WorkQue.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 *words_group;
char *key_array, *output_array;
long key_array_size, key_array_count, key_array_limit,
   new_key_array_size;
int filenumber;
char newword[15], hostname[256],filename[256];
int group_size, group_number; // size of the group of word to send by
   rank 0
int *Free; // An array that tells rank 0 which processor is free
int *Order;// An array that tells the rank 0 the order of the group of
   word that was sent

/* 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 );
```

```
52      count[i] = 0;
53    }
54
55    line = (char **) malloc( maxlines * sizeof( char * ) );
56    for( i = 0; i < maxlines; i++ ) {
57      line[i] = malloc( 2001 );
58    }
59
60    // Vairables below are for storing the output
61      key_array_count = 0;      // counter
62      key_array_limit =  99000000; // limit to increase array size: 99
          million
63      key_array_size  = 100000000; // size of key_array: 100 million ~ 95
           MB
64      new_key_array_size = key_array_size; // new array size (initail
          value is key_array_size)
65    key_array = malloc(sizeof(char)*key_array_size);
66    key_array[0] = '\0';
67
68    // The array below is for printing the output in the correct order
69    output_array = malloc(sizeof(char)*key_array_size);
70    output_array[0] = '\0';
71    // An array to group a list of keywords
72    words_group = malloc(sizeof(char)*1200);
73
74    // This is an array that tells rank 0 which processor is free
75    Free = (int *) malloc( nthreads * sizeof( int ) ); // 0 free, 1 busy
76    for (i = 0; i < nthreads; i++)
77    {
78      Free[i] = 0; //Initial values (all are free)
79    }
80
81    // This array will record the order of the group of word sent to each
          processor
82    Order = (int *) malloc( maxwords/group_size * sizeof( int ) );
83    for( i = 0; i < maxwords/group_size; i++ ) {
84      Order[i] = 0;
85    }
86
87  }
88
89
90  /* read_dict_words: (Read Dictianary words)
91  ---------------------------------------------*/
92  void read_dict_words()
93  {
94  // Read in the dictionary words
95    fd = fopen("/homes/dan/625/keywords.txt", "r" );
96    nwords = -1;
97    do {
98      err = fscanf( fd, "%[^\n]\n", word[++nwords] );
99    } while( err != EOF && nwords < maxwords );
100   fclose( fd );
101
102   //printf( "Read in %d words\n", nwords);
```

```
103 }
104
105
106 /* read_lines: (Read wiki lines)
107 ---------------------------------------------*/
108 void read_lines()
109 {
110 // Read in the lines from the data file
111   double nchars = 0;
112   fd = fopen( "/homes/dan/625/wiki_dump.txt", "r" );
113   nlines = -1;
114   do {
115     err = fscanf( fd, "%[^\n]\n", line[++nlines] );
116     if( line[nlines] != NULL ) nchars += (double) strlen( line[nlines]
        );
117   } while( err != EOF && nlines < maxlines);
118   fclose( fd );
119
120   //printf( "Read in %d lines averaging %.0lf chars/line\n", nlines,
       nchars / nlines);
121 }
122
123
124 /* reset_file: (create an output file)
125 ---------------------------------------------*/
126 void *reset_file()
127 {
128   snprintf(filename,250,"wiki-mpi-%d.out", filenumber);
129   fd = fopen(filename, "w" );
130   fclose( fd );
131 }
132
133
134 /* count_words: (count the words)
135 ---------------------------------------------*/
136 void *count_words(void *rank)
137 {
138
139   int i,k;
140   int myID =  *((int*) rank);
141
142   //Reset counter array:
143   for( i = 0; i < nwords; i++ ) {
144     count[i] = 0;
145   }
146
147   //Start counting and recording:
148   for( i = 0; i < nwords; i++ )
149   {
150     for( k = 0; k < maxlines; k++ )
151     {
152       if (count[i] < 100) // Look up to 100 occurance
153       {
154         if( strstr( line[k], word[i] ) != NULL )
155         {
```

```
156             if (count[i] == 0)
157             {
158               snprintf(newword,10,"%s:",word[i]);
159               strcat(key_array,newword);
160               key_array_count += strlen(newword);
161               //if (strstr( newword,"cico") != NULL) printf("found %d, %s
                      \n",myID, newword);
162             }
163             count[i]++;
164             snprintf(newword,10,"%d,",k);
165             strcat(key_array,newword);
166             key_array_count += strlen(newword);
167           }
168         }
169
170       }
171
172       if (count[i] != 0)
173       {
174         snprintf(newword,5,"\n");
175         strcat(key_array,newword);
176         key_array_count += strlen(newword);
177         }
178
179         // Checks if the size of the key_array has reached the limit
180       if (key_array_count > key_array_limit)
181       {
182         new_key_array_size += key_array_size; // incrase size
183         char* myrealloced_array = realloc(key_array, new_key_array_size *
                sizeof(char));
184         if (myrealloced_array) key_array = myrealloced_array;
185         key_array_count = 0; // reset counter
186       }
187
188     }
189   }
190
191
192   /* group_words: (Group words in to one array)
193   ------------------------------------------*/
194   void *group_words(int gnumber)
195   {
196     words_group[0] = '\0';
197     for (int i = gnumber*group_size; i < (gnumber+1)*group_size; ++i)
198     {
199       strcat(words_group,word[i]);
200       strcat(words_group,"\n");
201     }
202   }
203
204   /* split_words: (Split a group of words)
205   ------------------------------------------*/
206   void *split_words()
207   {
208     nwords = -1;
```

```
209   int word_size = 0;
210   do {
211     err = sscanf( words_group+word_size, "%[^\n]\n", word[++nwords]);
212     word_size += strlen(word[nwords]) + 1;
213   } while( err != EOF && nwords < group_size );
214
215 }
216
217 /* delet elements in an array
218 -----------------------------*/
219 void *remove_elements(char *array, int array_length, int index)
220 {
221    int i;
222    for(i = 0; i < array_length - index; i++)
223    {
224     array[i] = array[i+index];
225    }
226    /*
227    //or
228    for(i = 0; i < array_length ; i++)
229    {
230     if (i <array_length- index)   array[i] = array[i+index];
231     else    array[i] = '?';
232    }*/
233 }
234
235 /* dump_words: (write the output on file)
236 ---------------------------------------------*/
237 void *dump_words()
238 {
239   fd = fopen(filename, "a" );
240   sscanf( key_array, "%[^?]\n", output_array);
241   int results =fputs(output_array,fd);
242   fclose( fd );
243   remove_elements(key_array,strlen(key_array),strlen(output_array)+4);
244   memset(&output_array[0], 0, sizeof(output_array));
245 }
246
247
248 /*-------------------------------------------------
249             Main
250 -------------------------------------------------*/
251 int main(int argc, char* argv[])
252 {
253
254   int i, rc;
255   int numtasks, rank;
256   double tstart_init, tend_int, tstart_count, tend_count, tend_reduce;
257     struct rusage ru;
258
259   MPI_Status Status;
260     MPI_Request Request;
261   maxlines = 100000; // Default Value
262   filenumber = rand() %100000; // Default Value
263   if (argc >= 2){
```

```
264       maxlines = atol(argv[1]);
265       filenumber = atol(argv[2]);
266     }
267
268     rc = MPI_Init(&argc,&argv);
269     if (rc != MPI_SUCCESS){
270       printf ("Error starting MPI program. Terminating.\n");
271       MPI_Abort(MPI_COMM_WORLD, rc);
272     }
273
274       MPI_Comm_size(MPI_COMM_WORLD,&numtasks); // Number of cores
275       MPI_Comm_rank(MPI_COMM_WORLD,&rank);    // rank of each core
276
277     nthreads = numtasks;
278
279     gethostname(hostname,255);
280     if(rank == nthreads-1) reset_file(); // The last core will reset the
          output file
281     MPI_Bcast(filename, 256, MPI_CHAR, nthreads-1, MPI_COMM_WORLD);
282     tstart = myclock(); // Global Clock
283
284     // Initialization :
285     //-----------------
286
287     group_size = 100;
288     group_number = 0;
289     int flag = 0;
290     int flag2 = 0;
291     int ord = 0;
292       tstart_init = MPI_Wtime();  // Private Clock for each core
293     init_list(&rank);
294     if (rank != 0)read_lines();
295     else read_dict_words();
296     MPI_Barrier(MPI_COMM_WORLD);
297
298     //MPI_Bcast( array, 100, MPI_INT, root, comm);
299     //MPI_Send(const void *buf, int count, MPI_Datatype datatype, int
          dest, int tag, MPI_Comm comm)
300     //MPI_Isend(const void *buf, int count, MPI_Datatype datatype, int
          dest, int tag, MPI_Comm comm, MPI_Request *request)
301     //MPI_Recv(void *buf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Status *status)
302     //MPI_Irecv(void *buf, int count, MPI_Datatype datatype, int source,
          int tag, MPI_Comm comm, MPI_Request *request)
303     //MPI_IPROBE(source, tag, comm, flag, status)
304     //MPI_Cancel(MPI_Request *request)
305     //MPI_Abort(MPI_Comm comm, int errorcode)
306     //MPI_File_open(MPI_Comm comm, char *filename, int amode, MPI_Info
          info, MPI_File *fh)
307     while (group_number < maxwords/group_size)
308     {
309       // Sending keywords from rank 0 to all the other:
310       //-----------------------------------------------
311       if (rank == 0)
312       {
```

```
313         for (int receiver = 1; receiver < nthreads; receiver++)
314         {
315           if (Free[receiver] == 0)
316           {
317             group_words(group_number);
318             group_number += 1;
319             Free[receiver] = 1; // Means that the reciever is now busy
320             MPI_Isend(&Free[receiver], 1, MPI_INT, receiver, 1234,
                    MPI_COMM_WORLD, &Request);
321             Order[ord] = receiver;
322             ord++;
323             MPI_Isend(words_group, 1200, MPI_CHAR, receiver, 5678,
                    MPI_COMM_WORLD, &Request);
324           }
325           else
326           {
327             MPI_Iprobe(receiver, 4321, MPI_COMM_WORLD, &flag, &Status );
328             if (flag)
329             {
330               MPI_Irecv(&Free[receiver], 1, MPI_INT, receiver, 4321,
                      MPI_COMM_WORLD, &Request);
331             }
332           }
333         }
334       }
335
336       // Receiving  keywords from rank 0:
337       //-------------------------------------------------
338       else
339       {
340         MPI_Iprobe(0, 1234, MPI_COMM_WORLD, &flag, &Status );
341         if (flag)
342         {
343           MPI_Irecv(&Free[rank], 1, MPI_INT, 0, 1234, MPI_COMM_WORLD, &
                  Request);
344           if (Free[rank] == 1)
345           {
346             while(!flag2)
347                 {
348                   MPI_Iprobe( 0, 5678, MPI_COMM_WORLD, &flag2, &Status );
349                 }
350             MPI_Irecv(words_group, 1200, MPI_CHAR, 0, 5678,
                    MPI_COMM_WORLD, &Request);
351             split_words();
352             count_words(&rank); // Counting
353             strcat(key_array,"???\n");
354             flag2 = 0;
355             flag = 0;
356             Free[rank] = 0; // Means that the reciever is now free
357             MPI_Isend(&Free[rank], 1, MPI_INT,0,4321,MPI_COMM_WORLD, &
                    Request);  // Tell rank 0 you r free
358           }
359         }
360         MPI_Iprobe(0, 9999, MPI_COMM_WORLD, &flag, &Status );
361         if (flag)
```

```
362         {
363           MPI_Irecv(&group_number, 1, MPI_INT, 0, 9999, MPI_COMM_WORLD, &
                Request);
364         }
365       }
366     }
367
368     // Exiting the counting Processs:
369     //-------------------------------------------------
370     if(rank==0)
371     {
372       for (int receiver = 1; receiver < nthreads; receiver++)
373       {
374         MPI_Isend(&group_number, 1, MPI_INT, receiver,9999,
              MPI_COMM_WORLD, &Request);
375       }
376     }
377     printf("Done rank %d\n",rank);
378     MPI_Barrier(MPI_COMM_WORLD);
379     // Probing for messages from the counting Processs:
380     //-------------------------------------------------
381     i = 1;
382     if(rank==0)
383     {
384       while(i == 1)
385       {
386         i = 0;
387         for (int receiver = 1; receiver < nthreads; receiver++)
388         {
389           MPI_Iprobe(receiver, 4321, MPI_COMM_WORLD, &flag, &Status );
390           if (flag)
391             {
392               printf("Message 4321 was not recieved from rank %d \n",
                    receiver);
393               MPI_Irecv(&Free[receiver], 1, MPI_INT, receiver, 4321,
                    MPI_COMM_WORLD, &Request);
394               i = 1;
395             }
396         }
397       }
398     }
399     else
400     {
401       MPI_Iprobe(0, 1234, MPI_COMM_WORLD, &flag, &Status );
402       if (flag)  printf("Message 1234 not recieved, rank %d,1,%d\n",rank,
            flag);
403       MPI_Iprobe(0, 9999, MPI_COMM_WORLD, &flag, &Status );
404       if (flag)  printf("Message 9999 not recieved, rank %d,1,%d\n",rank,
            flag);
405     }
406     MPI_Barrier(MPI_COMM_WORLD);
407     // Sending output to rank 0:
408     //-------------------------------------------------
409
410     MPI_Bcast(Order,maxwords/group_size, MPI_INT, 0, MPI_COMM_WORLD);
```

```
411     MPI_Barrier(MPI_COMM_WORLD);
412
413     // Priniting final data output to rank 0:
414     //-----------------------------------------------
415
416     for(i = 0; i < maxwords/group_size;i++)
417     {
418       if (rank == Order[i])
419       {
420       dump_words();
421       }
422       MPI_Barrier(MPI_COMM_WORLD);
423     }
424     getrusage(RUSAGE_SELF, &ru);
425       long MEMORY_USAGE = ru.ru_maxrss;   // Memory usage in Kb
426     printf(" rank %d, hostaname %s, size %d, memory %ld Kb\n",
427                 rank, hostname, nthreads,  MEMORY_USAGE);
428     fflush(stdout);
429
430     if ( rank == 0 ) {
431     ttotal = myclock() - tstart;
432     printf("version %d, cores %d, total time %lf seconds, words %d, lines
          %d\n",
433         version, nthreads, ttotal, nwords, maxlines);
434     }
435     MPI_Finalize();
436     return 0;
437 }
```

Listing 4: Bash script

```
1  #!/bin/bash
2  #$ -l mem=1G
3  #$ -l h_rt=24:00:00
4  #$ -l killable
5  #$ -cwd
6  #$ -q \*@@elves
7  #$ -pe mpi-fill 16
8
9  for i in 1000000 800000 600000 400000 200000
10 do
11 mpirun -np 16 /homes/mcheikh/CIS_625/hw3/MPI_V2.out $i 264
12 hostname
13 echo -e "---Done---\n"
14 done
```