

CS 442 Introduction to Parallel Processing

Project 1: MPI Sieve of Eratosthenes

Matt Bohnsack and Erik Erhardt

April 1, 2006

Contents

1	Executive Summary	2
2	Problem Description	3
3	Data Generation	4
4	Materials and Methods	5
5	Results	6
6	Conclusions	7
A	Appendix	8

List of Figures

List of Tables

1 Executive Summary

2 Problem Description

The Sieve of Eratosthenes is a simple, ancient algorithm for finding all prime numbers up to a specified integer.

First, write three things :

- * The highest factor to check for. It is the square root of the last number you care about, rounded down.
- * A list of primes you've found. It starts empty.
- * A list of numbers that remain to be checked. It starts out filled with all integers from two up to the last number you care about.

Then, repeat the following until you've checked all you need :

- * Remove the first remaining number to the primes list.
- * Delete anything that's a multiple of the number you just removed.

Finally, remove all remaining numbers to the primes list.

3 Data Generation

4 Materials and Methods

5 Results

6 Conclusions

A Appendix

Listing 1: MPI_primes.c

```

1  /*
2  *   MPI Sieve of Eratosthenes
3  */
4
5
6  /* which mpicc --> /usr/parallel/mpich-gm.pgi/bin/mpicc (myrinet)
7
8  ssh erike@loslobos.alliance.unm.edu
9
10 mpicc MPI_primes.c -o MPI_primes
11 mpirun -np 8 -machinefile $PBS_NODEFILE ./MPI_primes -v -n 1000
12
13 qsub -I -l nodes=4:ppn=2,walltime=2:00:00
14 mpirun ./MPI_primes -n 1000
15
16 setenv MPI_np `cat $PBS_NODEFILE | wc -l`
17 mpirun -np $MPI_np ./MPI_primes -n 1000
18
19
20
21 There were a couple of issues I worked through.
22 * 1 - when rank 0 is done sending numbers, rank 1 still has to listen to the last rank
23 * 2 - each rank, when seeing num=-1 on the last loop, still has to pass the -1 on to the next rank
24 * 2a - but after passing the -1 on, it can complete.
25 * 3 - I added a few switches to facilitate all of this. It is possible to make this more efficient, but it currently works
26
27
28 Matt's Notes:
29
30 * You can probably just do the following instead of fflush()ing all the time,
31   if you always want unbuffered stdout: setvbuf(stdout, NULL, _IONBF, 0);
32
33
34
35 3/29/2006 12:54PM
36 ** check for CAPS COMMENTS
37   DONE iota is sending block
38   people receive block
39   people sending block
40   create the SIZE variable indicating the size of the nums remaining
41   loop for killing
42
43
44 */
45
46 #include <mpi.h>
47 #include <time.h>
48 #include <math.h>
49 #include <errno.h>
50 #include <stdio.h>
51 #include <ctype.h>
52 #include <stdlib.h>
53 #include <unistd.h>
54 #include <string.h>
55 #include <sys/wait.h>
56 #include <sys/time.h>
57 #include <sys/stat.h>
58 #include <sys/types.h>
59 #include <sys/resource.h>
60
61 /***** Globals *****/
62 struct program_options
63 {
64     long n;
65     int block_size;
66     int init_first_three_primes;
67     int debug_level;
68 } opts;
69 struct program_options opts = {0, 1, 0, 0};
70
71 /***** Function Prototypes *****/
72 void iota(int rank, long *num_array, long *nums);
73 void recv_from(int from, long *num_array, int *loop, int *nums_size, long *nums);
74 void send_to(int to, long *num_array, int *loop, int *nums_size, long *nums, int increment_loop);
75 void display_primes(long int* array, int n_primes, int buff_size, int numprocs);
76 void parse_command_line_args(int argc, char *argv[]);
77 int compare longs(const void *a, const void *b);
78 void disp_nums(char *string, int rank, long *nums, int nums_size, int index);
79
80 /***** Main *****/
81 int main (int argc, char *argv[])
82 {

```

```

83     int i = 0;
84     long int n_primes = 0;
85     int k = 0;
86     long int *primes = NULL;
87     long int *all_primes = NULL;
88     int kill_sw = 0;
89     int added_prime_sw = 0;
90     int checking_complete_sw = 0;
91     int rank0_done_sw = 0;
92     int last_prime_index = 0;
93     int indy_gather_buffer_size = 0;
94     int numprocs = 0;
95     int rank = 0;
96     long *num_array = NULL;
97     long *nums = NULL;
98     int loop=0;
99     int nums_size=1;
100    int i_live = -1;
101
102    /* communications */
103    MPI_Status status_sreq, status_rreq;
104    MPI_Request sreq, rreq;
105    int ierr, tag=0;
106
107    /* printf("arguments 1: %d\n", argc); */
108    MPI_Init(&argc, &argv);
109    /* printf("arguments 2: %d\n", argc); */
110    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
111    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
112
113    parse_command_line_args(argc, argv);
114
115    primes = (long int *)malloc(opts.n * sizeof(long int));
116    if(primes == NULL){fprintf(stderr, "Out of memory!\n");exit( EXIT_FAILURE );}
117
118    num_array = (long int *)malloc((opts.block_size+2) * sizeof(long int));
119    if(num_array == NULL){fprintf(stderr, "Out of memory!\n");exit( EXIT_FAILURE );}
120
121    nums = &num_array[2];
122
123    /* setup initial list of primes */
124    if (rank==1) {
125        primes[0] = 2;          /* rank 1 starts with primes array with 2 */
126        last_prime_index = 0;
127    } else {
128        last_prime_index = -1; /* ranks > 1 start with empty primes array */
129    }
130
131    MPI_Barrier(MPI_COMM_WORLD);
132    if(opts.debug_level){printf("There are %d processes and I am rank #%d\n", numprocs, rank);fflush(stdout);}
133    MPI_Barrier(MPI_COMM_WORLD);
134
135
136    if (rank==0) /* rank 0 generates integers 2..n */
137    {
138        iota(rank, num_array, nums);
139    }
140    else /* positive ranks identify primes from numbers from rank 0 */
141    {
142        int flag=0; /* used in MPI_Iprobe */
143
144        do{
145            kill_sw = 0;
146            added_prime_sw = 0;
147            checking_complete_sw = 0;
148            if(rank==1) /* can receive from rank 0 and from rank (numprocs-1) */
149            {
150                /* rank 1 probes the last rank to see if it has anything to send */
151                ierr = MPI_Iprobe(numprocs-1, tag, MPI_COMM_WORLD, &flag, &status_rreq); if (ierr != MPI_SUCCESS) {printf("M
152                if(flag || rank0_done_sw) /* message ready to be received from rank (numprocs-1) */
153                {
154                    recv_from(numprocs-1, num_array, &loop, &nums_size, nums);
155                }
156                else /* receive from rank 0 */
157                {
158                    recv_from(0, num_array, &loop, &nums_size, nums);
159                    if (nums[0] == -1)
160                    {
161                        rank0_done_sw = 1;
162                    }
163                }
164            }
165            else /* receive from rank-1 */
166            {
167                recv_from(rank-1, num_array, &loop, &nums_size, nums);
168            }
169

```

```

170
171     if(loop > last_prime_index) /* add seived number to list of primes */
172     {
173         if(nums[0] != -1)
174         {
175             last_prime_index++;
176             primes[last_prime_index] = nums[0];
177
178             for(i = 0; i < nums_size - 1; i++) /* shift the remaining values in nums up by one index */
179             {
180                 nums[i] = nums[i+1];
181             }
182             nums_size--;
183         }
184         else /* nums[0] == -1, so done checking for primes */
185         {
186             checking_complete_sw = 1;
187         }
188     }
189
190     if(!checking_complete_sw){
191
192         /* LOOP THROUGH THE NUMS ARRAY TO KILL THEM */
193         /* CRUNCH THE REMAINING LIST BY nums[i_live]=nums[ind] */
194         /* UPDATE SIZE */
195         if(opts.debug_level>1){disp_nums("Before Loop", rank, nums, nums_size, 0);};
196         if(opts.debug_level>1){printf("Rank %d nums_size = %d, prime=%d\n", rank, nums_size, primes[loop]);};
197         i_live = -1;
198         for(i=0; i < nums_size; i++) /* loop through the nums array, keeping the relatively prime values */
199         {
200             if(nums[i] % primes[loop]) /* relatively prime - so we retain the number */
201             {
202                 i_live++;
203                 nums[i_live] = nums[i]; /* retain number by moving up to the i_live index */
204                 if(opts.debug_level>1){disp_nums("After Shuffle Step", rank, nums, nums_size, i);};
205             }
206         }
207
208         nums_size = i_live+1;
209         if(nums_size == 0) /* all nums killed, nothing to send to next process */
210         {
211             kill_sw = 1;
212         }
213         if(opts.debug_level>1){disp_nums("After Loop", rank, nums, nums_size, 0);};
214         if(opts.debug_level>1){printf("Rank %d nums_size = %d\n", rank, nums_size);};
215     }
216
217     if (!kill_sw || checking_complete_sw) /* the number has not been killed, send on to next rank */
218     {
219         if (rank < numprocs-1) /* non last rank sends to next rank */
220         {
221             send_to(rank+1, num_array, &loop, &nums_size, nums, 0);
222         }
223         else /* last rank sends to rank 1 */
224         {
225             send_to(1, num_array, &loop, &nums_size, nums, 1);
226         }
227     }
228
229     } while (!checking_complete_sw);
230 }
231
232 if(opts.debug_level){printf("FINISHED rank %d\n", rank);};
233
234 /* Each rank prints the primes it found */
235 /* MPI_Barrier(MPI_COMM_WORLD); */
236 if(opts.debug_level && rank !=0)
237 {
238     printf("I found these primes (and I am very proud!):\n");fflush(stdout);
239     for(k=0; k <= last_prime_index; k++)
240     {
241         printf("rank: %d prime: %d\n", rank, primes[k]);fflush(stdout);
242     }
243 }
244
245 /* MPI_Barrier(MPI_COMM_WORLD); */
246
247 /* Tell rank 0 how many primes total */
248 last_prime_index++;
249 MPI_Reduce(&last_prime_index, &n_primes, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
250 if (rank==0) {if(opts.debug_level){printf("rank: %d n_prime = %d\n", rank, n_primes);fflush(stdout);};}
251
252 /* rank 1 has the most primes so tells rank 0 indy_gather_buffer_size, to use as MPI_Gather buffer size */
253 if (rank==1) {
254     ierr = MPI_Isend(&last_prime_index, 1, MPI_INT, 0, tag, MPI_COMM_WORLD, &sreq); if (ierr != MPI_SUCCESS) {printf("MPI_Isend ERROR status %d.\n", ierr);}
255     ierr = MPI_Wait(&sreq, MPI_STATUSES_IGNORE); if (ierr != MPI_SUCCESS) {printf("MPI_Wait ERROR status %d.\n", ierr);}
256 }

```

```

257  /* rank 1 tells rank 0 indy_gather_buffer_size, the MPI_Gather buffer size */
258  if (rank==0) {
259      ierr = MPI_Irecv(&indy_gather_buffer_size, 1, MPI_INT, 1, tag, MPI_COMM_WORLD, &rreq); if (ierr != MPI_SUCCESS) {pr
260      ierr = MPI_Wait(&rreq, MPI_STATUSES_IGNORE); if (ierr != MPI_SUCCESS) {printf("MPI_Wait ERROR status %d.\n", ierr);
261  }
262
263  /* rank 1 tells all other ranks the indy_gather_buffer_size */
264  if (rank==1) { indy_gather_buffer_size = last_prime_index;}
265  ierr = MPI_Bcast(&indy_gather_buffer_size, 1, MPI_INT, 1, MPI_COMM_WORLD); if (ierr != MPI_SUCCESS) {printf("MPI_Isend E
266
267  /* rank 0 allocate array of primes */
268  if (rank==0) {
269      all_primes = (long int *)malloc(indy_gather_buffer_size * numprocs * sizeof(long int));
270  }
271  /* MPI_Barrier(MPI_COMM_WORLD); */
272
273  /* rank 0 gathers array of primes from other ranks */
274  MPI_Gather(primes, indy_gather_buffer_size, MPI_LONG, all_primes, indy_gather_buffer_size, MPI_LONG, 0, MPI_COMM_WORLD);
275
276  /* Need to sort the all_primes array, and start the printing at the first non 0. */
277
278  if(opts.debug_level && rank == 0)
279  {
280      display_primes(all_primes, n_primes, indy_gather_buffer_size, numprocs);
281  }
282
283  MPI_Finalize();
284  return 0;
285  }
286  /* END of main() */
287
288
289  /***** Functions *****/
290
291  void iota(int rank, long *num_array, long *nums)
292  {
293      long num=2;
294      int loop=0, nums_size=0;
295      MPI_Status status_sreq, status_rreq;
296      MPI_Request sreq, rreq;
297      int ierr, tag=0;
298      int iblock=0, block=0, nblocks=0;
299
300
301      if(opts.debug_level){printf("block size= %d ", opts.block_size);};
302
303      num_array[0] = loop; /* first array element has loop which will always be 0 from iota (rank=0) */
304      nums_size = opts.block_size;
305      num_array[1] = nums_size;
306
307      nblocks = (int)(opts.n/opts.block_size); /* toss away remainder for now... *****/
308
309      for(block = 0; block < nblocks ; block++) /* create candidate numbers */
310      {
311          for(iblock = 0; iblock < opts.block_size ; iblock++, num++) /* create blocks */
312          {
313              num_array[iblock+2] = num; /* block*opts.block_size + iblock; */
314          }
315          send_to(1, num_array, &loop, &nums_size, nums, 0);
316      }
317      num = -1; /* end of numbers */
318      nums_size = 1;
319      num_array[1] = nums_size;
320      num_array[2] = num;
321      send_to(1, num_array, &loop, &nums_size, nums, 0);
322  }
323
324  void recv_from(int from, /* rank receiving message from */
325                long *num_array, /* block of loop/nums_size/nums being received */
326                int *loop, /* loop number from num_array[0] */
327                int *nums_size, /* the number of remaining candidate numbers in nums from num_array[1] */
328                long *nums /* the list of remaining candidate numbers from num_array[2-end] */
329                )
330  {
331      MPI_Status status_sreq, status_rreq;
332      MPI_Request sreq, rreq;
333      int ierr, tag=0;
334
335      ierr = MPI_Irecv(num_array, opts.block_size+2, MPI_LONG, from, tag, MPI_COMM_WORLD, &rreq); if (ierr != MPI_SUCCESS) {pr
336      ierr = MPI_Wait(&rreq, MPI_STATUSES_IGNORE); if (ierr != MPI_SUCCESS) {printf("MPI_Wait ERROR status %d.\n", ierr); ierr
337      *loop=num_array[0];
338      *nums_size=num_array[1];
339      nums=&num_array[2];
340
341  }
342
343  void send_to(int to, /* rank receiving message from */

```

```

344         long *num_array, /* block of loop/nums_size/nums being received */
345         int *loop, /* loop number from num_array[0] */
346         int *nums_size, /* the number of remaining candidate numbers in nums from num_array[1] */
347         long *nums, /* [NOT NEEDED] the list of remaining candidate numbers from num_array[2-end] */
348         int increment_loop /* a flag 0=don't increment loop, 1=do (when last rank passes to rank 1)*/
349     )
350 {
351     MPI_Status status_sreq, status_rreq;
352     MPI_Request sreq, rreq;
353     int ierr, tag=0;
354
355     num_array[0] = (*loop)+increment_loop;
356     num_array[1] = *nums_size;
357     /* Note: don't need to assign nums to num_array[2] since shared address space */
358     ierr = MPI_Isend(num_array, opts.block_size+2, MPI_LONG, to, tag, MPI_COMM_WORLD, &sreq); if (ierr != MPI_SUCCESS) {prin
359     ierr = MPI_Wait(&sreq, MPI_STATUSES_IGNORE); if (ierr != MPI_SUCCESS) {printf("MPI_Wait ERROR status %d.\n", ierr); ierr
360 }
361
362
363 void display_primes(long int* array, int n_primes, int buff_size, int numprocs)
364 {
365     int i = 0;
366     int array_size = buff_size * numprocs;
367
368     qsort(array, array_size, sizeof(long int), compare_longs);
369
370     for( i = 0; i < n_primes; i++)
371     {
372         printf("Prime #%d is: %d\n", i+1, array[i+(array_size-n_primes)]); fflush(stdout); /* to start with first non-zero pr
373     }
374 }
375
376 int compare_longs(const void *a, const void *b)
377 {
378     const long int *da = (const long int *) a;
379     const long int *db = (const long int *) b;
380
381     return (*da > *db) - (*da < *db);
382 }
383
384 void parse_command_line_args(int argc, char *argv[])
385 {
386     int c = 0;
387
388     while( (c = getopt( argc, argv, "n:ovb:" )) != EOF )
389     {
390         switch( c )
391         {
392             case 'n': opts.n = strtoul( optarg, (char **)NULL, 10 );
393             break;
394
395             case 'b': opts.block_size = strtoul( optarg, (char **)NULL, 10 );
396             break;
397
398             case 'o': opts.init_first_three_primes = 1;
399             break;
400
401             case 'v': opts.debug_level++;
402             break;
403         }
404     }
405 }
406
407 void disp_nums(char * string, int rank, long *nums, int nums_size, int index)
408 {
409     int i = 0;
410     printf("Rank %d nums %s on i=%d\n", rank, string, index); fflush(stdout);
411     for( i = 0 ; i < nums_size; i++)
412     {
413         printf("Rank %d nums [%d]: %d\n", rank, i, nums[i]); fflush(stdout);
414     }
415 }

```