Parallel Quicksort algorithm

Programming of Parallel Computers, 10c
Uppsala, 2016

Author

Alessandro Piccolo
Uppsala University

1 Introduction

The problem at hand is to efficiently sort a sequence of double elements incrementally on multiple processors. The purpose of this project is to implement an algorithm on parallel processors. The solution in presented in this report is to use a quicksort algorithm in parallel. Quicksort is a comparison sort and the compared element, the so called pivot element. The pivot element in this report has been chosen as the last element in the array. After the pivot has been chosen the array is divided into two smaller arrays, elements lower than the pivot to the left and elements higher than the pivot to the right. This procedure is done recursively to sort the array.

The parallelization part of the program is to divide the array into smaller chunks and do a quicksort locally. After the elements are sorted the processors need to exchange parts of the sub array in-between processors such that half of the processors have elements lower than the chosen pivot and the other half have elements higher than the pivot. This time the pivot is chosen to be the mean value of all the sub arrays in that specific group. As a final step the processor group is divided into two parts as above, lesseränd greaterprocessors. This procedure is done recursively, for simplification there is a pseudo algorithm below.

Algorithm 1 Parallelization algorithm of Quicksort

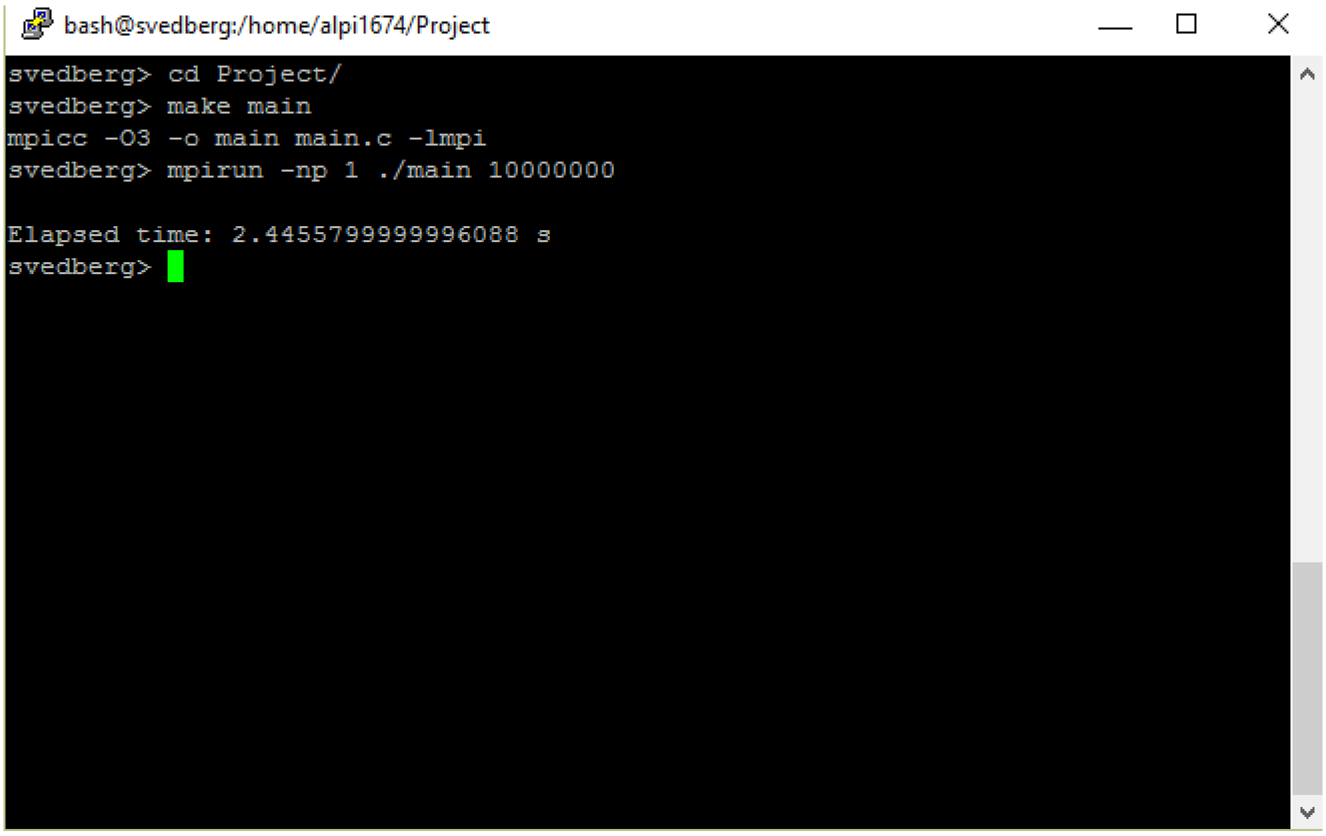
```
1: Divide array into equally sized sub arrays
2: procedure QUICKSORT PARALLEL($data$, $length$, $comm$)
3:     if $nprocs = 1$ then
4:         return
5:     end if
6:     Find pivot as mean value of all processor in group comm
7:     Bcast calculated pivot from root processor
8:     Split local sub array into two halves, lower and greater than pivot
9:     if $rank < nprocs/2$ then
10:        Send greater half of sub array to processor with rank $rank + procs/2$
11:        Receive lower half of sub array from processor with rank $rank + procs/2$
12:    else
13:        Receive higher half of sub array from processor with rank $rank - procs/2$
14:        Send lower half of sub array to processor with rank $rank - procs/2$
15:    end if
16:    Merge received array with kept array
17:    Split communication into two halves
18:    Call procedure recursively
19: end procedure
```

Algorithm 2 Calculating the center of mass (CoM) and total mass

```
 1: Returns structure returnV
 2: procedure CENTRE_OF_MASS($treeNode * node$)
 3:     if (node contains one particle) then
 4:         Save particle coordinates and mass in node
 5:         Save particle coordinates and mass into returnV
 6:         return returnV
 7:     else
 8:         for i = 1 to 4 do
 9:             $CoM_{sum} += child_i.coord \cdot child_i.mass$
10:             $mass_{sum} += child_i.mass$
11:         end for
12:         Save particle coordinates and mass in node
13:         Save particle coordinates and mass into returnV
14:     end if
15:     return returnV
16: end procedure
```

2 Results

The following speedup plots were achieved on Solaris x86 64 with login node on svedberg.it.uu.se. The following data points are the mean value of five runs. The plots used 1, 2, 4 and 8 processors and a randomized 10 million double element vector. A normal run would look like this,



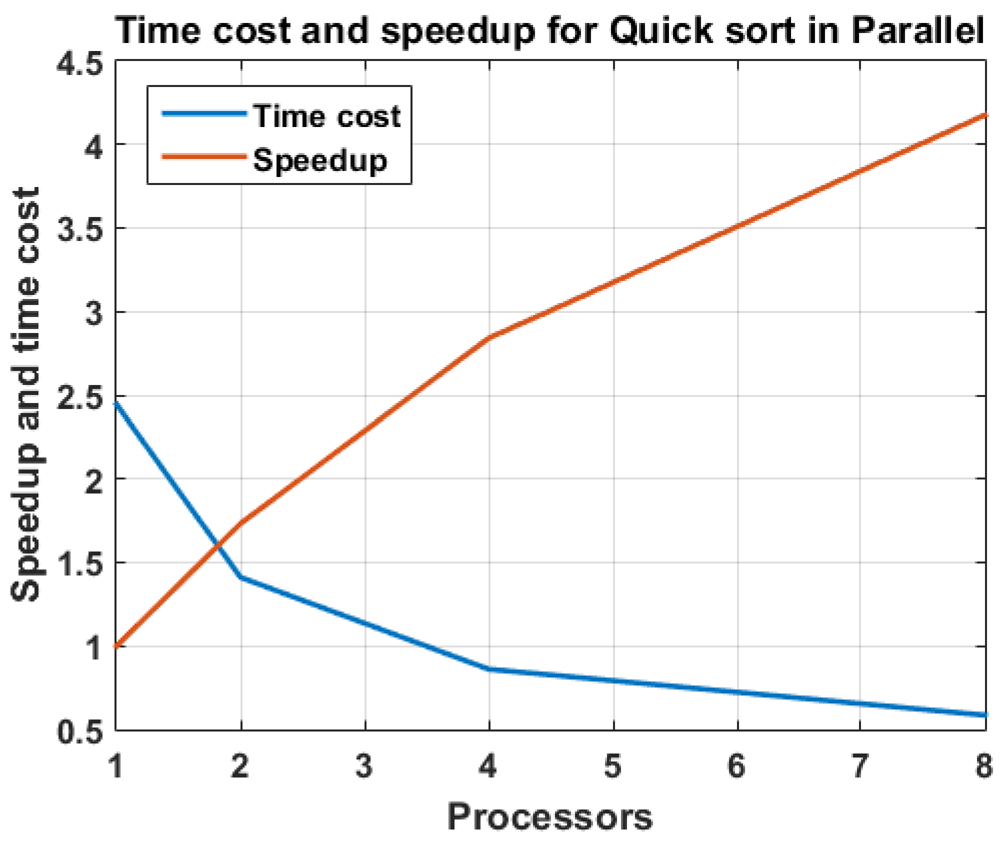
bash@svedberg:/home/alpi1674/Project

```
svedberg> cd Project/
svedberg> make main
mpicc -O3 -o main main.c -lmpi
svedberg> mpirun -np 1 ./main 10000000

Elapsed time: 2.4455799999996088 s
svedberg> 
```

Figur 1: Shows the program beeing executed with only one processor on a vector of 10 million elements.

The run from figure 1 together with more data points gives us figure 2.



Figur 2: Shows the speedup (red line) versus the time cost (blue line) on the y-axis over the number of processors on a vector of 10 million elements.

3 Conclusions

From figure 2 one can see that the speedup almost has a linear speedup between one and four processors and from one to eight the speedup slope drops slightly. The more processors that are added the more communication is needed in-between the processors hence there will be a lower speedup. The algorithm could be more effective regarding some of the functions such as the merge and quick sort in serial functions.

Appendices

A speedupVsTimecost.m

```
% speedup
clear all; clc; close all;

% #threads
processors = [1 2 4 8];

% Experimental data

% time; 1 processor; 2 processor; 4 processor; processor 8: processor 16;
time = [2.4455799999996088 1.4056020000134595 0.8547910000197589 0.5539709999575280;
        2.4428740000003017 1.4161969999549910 0.8663079999969341 0.5506129999994300;
        2.4480209999601357 1.4047319999663159 0.8514680000371300 0.5663649999769405;
        2.4452739999978803 1.4123290000134148 0.8563059999723919 0.6018219999969006;
        2.4454329999862239  1.4169799999799579  0.8768750000162981  0.6547780000255443];
time_mean = mean(time);

% Calculating speedup, S = Tinit/Tnew
speedup = time_mean(1)./time_mean(:);

figure
plot(processors,time_mean,processors,speedup, 'LineWidth',2)
set(gca,'FontSize',13, 'FontWeight', 'bold');
title(['Time cost and speedup for Quick sort in Parallel']);
ylabel('Speedup and time cost') % label left y-axis
xlabel('Processors') % label x-axis
legend('Time cost','Speedup')
grid on
```

B main.c

```
/*************************************************************************
 * Parallell quick sort algorithm
 * By Alessandro Piccolo
 *************************************************************************/

#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

void quick_sort_serial(double *vector, int a, int b);
void quick_sort_par(double *data, int length, MPI_Comm comm, int *last_length);
int search_split_point(double *data, int length, double pivot);
void merge(double *d1, int len1, double *d2, int len2, double *data_sort);
void copyArray(double *d1, double *d2, int start, int length);
void swap(double *vector, double temp, int i, int j);
double findMean(double *vector, int length);
int dbgSorted(double* data, int length);
void print_vector(double *vector, int length);
void print_vector_int(int* vector, int length);

int main(int argc, char *argv[]) {
  int rank;                                   /* Id of processors          */
  int nprocs;                                 /* Number of processor       */
  int n = atoi(argv[1]);                      /* n vector long             */
  int root = 0, i;
  double pivot, time_init, time_end;
  double *data, *data_sub, *data_sorted;
  int *last_length, *recieve_counts, *receive_displacements;
  last_length = (int *)malloc(1*sizeof(int));

  MPI_Init(&argc, &argv);                     /* Initialize MPI            */
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);     /* Get the number of processors */
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);       /* Get my number             */

  int elements_per_proc = n/nprocs;           /* Elements per processor    */

  /* Generate random numbers, pivot and start time */
  if (rank == 0) {
    if (n <= 1) {
      printf("Already sorted, only one element");
      return 1;
    }
    data = (double *)malloc(n*sizeof(double));
    data_sorted = (double *)malloc(n*sizeof(double));
    recieve_counts = (int *)malloc(nprocs*sizeof(int));
    receive_displacements = (int *)malloc(nprocs*sizeof(int));
    srand(time(NULL));
    for (i = 0; i < n; i++) {
        data[i] = drand48();
    }
    time_init = MPI_Wtime();
  }

  /* Scatter chunk of data into smaller data_sub vectors */
  data_sub = (double *)malloc(n*sizeof(double));
  MPI_Scatter(data, elements_per_proc, MPI_DOUBLE, data_sub,
              elements_per_proc, MPI_DOUBLE, root, MPI_COMM_WORLD);

  /* Quick sort in serial followed by parallel quick sort algorithm */
  quick_sort_serial(data_sub, 0, elements_per_proc-1);
  quick_sort_par(data_sub, elements_per_proc, MPI_COMM_WORLD, last_length);
  elements_per_proc = *last_length;

  /* Number of elements in each processor */
  MPI_Gather(&elements_per_proc, 1, MPI_INT, recieve_counts, 1,
             MPI_INT, root, MPI_COMM_WORLD);

  /* Fill displacement vector for gatherv function */
  if (rank == 0) {
    int index = 0; receive_displacements[0] = index;
    for (i = 1; i < nprocs; i++) {
      index = index + recieve_counts[i-1];
      receive_displacements[i] = index;
    }
  }

 /* Gather all sub vectors into one large sorted vector  */
  MPI_Gatherv(data_sub, elements_per_proc, MPI_DOUBLE, data_sorted,
      recieve_counts, receive_displacements, MPI_DOUBLE, root, MPI_COMM_WORLD);

  /* End time and free datatypes */
  if (rank == 0) {
    time_end = MPI_Wtime() - time_init;
    printf("\nElapsed time: %.16f s\n", time_end);

    if (!dbgSorted(data_sorted, n)) {
       printf("Error not sorted \n");
    }
  }
  free(last_length);
  free(data);
  free(data_sub);
  free(data_sorted);
  free(receive_displacements);
  free(recieve_counts);

  MPI_Finalize();                              /* Shut up and clean down MPI   */
  return 0;
}

/*
 * Sorts a vector with recursive quick sort algorithm, a and b defines
 * the interval of what to sort. Variable b is also the index of our
 * pivot element.
 */
void quick_sort_serial(double *vector, int a, int b) {
  double pivot;
  int i, j;

  if(a < b) {
    pivot = vector[b];
    i = a;
    j = b;

    while(i < j) {
      while(vector[i] < pivot) {
        i++;
      }
      while(vector[j] >= pivot) {
        j--;
      }
      if(i < j) {
        swap(vector, vector[i], i, j);
      }
    }
    swap(vector, pivot, b, i);

    quick_sort_serial(vector, a, i-1);
    quick_sort_serial(vector, i+1, b);
  }
}

/*
 * Quick sort parallel, length is the number of elements in
 * ata last_length is pointer to the number of elements in
 * the last recursive functions data vector.
 */
void quick_sort_par(double *data, int length, MPI_Comm comm, int *last_length){
  int nprocs, rank, pivot_index, number_amount, length_keep, i;
  double pivot, mean_local[2] = {0.0,length}, mean_global[2]={0.0,0.0};
  double *data_recieve, *data_keep;
  MPI_Status status;
  MPI_Comm new_comm;
  MPI_Comm_size(comm, &nprocs);
  MPI_Comm_rank(comm, &rank);

  if (nprocs == 1) { /* We are done, do not have to proceed */
    *last_length = length;
    return;
  }

  /* Calculate pivot as mean value across all processors */
  for (i = 0; i < length; i++) { /* Sum the numbers locally */
    mean_local[0] = mean_local[0] + data[i];
  }
  MPI_Reduce(&mean_local, &mean_global, 2, MPI_DOUBLE, MPI_SUM, 0, comm);
  if (rank == 0) {
    pivot = mean_global[0]/mean_global[1] ; /* Pivot is meanvalue */
  }
  MPI_Bcast(&pivot, 1, MPI_DOUBLE, 0, comm);

  /* Find local index where to split the vector into lesser and greater half */
  pivot_index = search_split_point(data, length, pivot);

  /* Exchange halves */
  if (rank < nprocs/2) {
    /* Send elem higher than pivot from left hand side processor */
    MPI_Send(data + (pivot_index + 1), (length - 1) - pivot_index, MPI_DOUBLE,
             rank + nprocs/2, 00, comm);
    /* Recieve elem lower than pivot from right hand side processor */
    MPI_Probe(rank + nprocs/2, 11, comm, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &number_amount);
    data_recieve = (double *)malloc(number_amount*sizeof(double));
    MPI_Recv(data_recieve, number_amount, MPI_DOUBLE, rank + nprocs/2,
             11, comm, MPI_STATUS_IGNORE);
  }
  else {
    /* Recieve elem higher than pivot from left hand side processor */
    MPI_Probe(rank - nprocs/2, 00, comm, &status);
    MPI_Get_count(&status, MPI_DOUBLE, &number_amount);
    data_recieve = (double *)malloc(number_amount*sizeof(double));
    MPI_Recv(data_recieve, number_amount, MPI_DOUBLE,
             rank - nprocs/2, 00, comm, MPI_STATUS_IGNORE);
    /* Send elem lower than pivot from right hand side processor */
    MPI_Send(data, pivot_index + 1, MPI_DOUBLE, rank - nprocs/2, 11, comm);
  }

  /* Create new array data to be kept, and to be merged with data recieved */
  if (rank < nprocs/2) {
    length_keep = pivot_index + 1;
    data_keep = (double *)malloc(length_keep*sizeof(double));
    copyArray(data, data_keep, 0, length_keep);
  }
  else {
    length_keep = (length - 1) - pivot_index;
    data_keep = (double *)malloc(length_keep*sizeof(double));
    copyArray(data, data_keep, pivot_index+1, length_keep);
  }

  /* Merge data into one */
  merge(data_recieve, number_amount, data_keep, length_keep, data);

  /* Split communicator into two parts, left and right */
  int color = rank/(nprocs/2);
  MPI_Comm_split(comm, color, rank, &new_comm);

  quick_sort_par(data, length_keep + number_amount, new_comm, last_length);
}

/*
 * Search for split point index. Returns the pivot index, [0,pivot_index]
 * are considered as the lower half <= pivot value.
 */
int search_split_point(double *data, int length, double pivot) {
  int i = 0;
  for (i = 0; i < length; i++) {
    if (data[i] > pivot) {
      return i-1;
    }
  }
  return i-1; /* Zero entry returns -1 */
}

/*
 * Merge two sorted arrays into one Sort d1 and d2 into data_sort
 * d1 and d2 have to be already sorted incrementally.
 */
void merge(double *d1, int len1, double *d2, int len2, double *data_sort) {
  int i = 0, j = 0, index = 0;

  while(i < len1 && j < len2) {
    if (d1[i] < d2[j]) {
      data_sort[index] = d1[i];
      i++;
    }
    else {
      data_sort[index] = d2[j];
      j++;
    }
    index++;
  }
  if (i >= len1) {
    while (j < len2) {
      data_sort[index] = d2[j];
      j++;
      index++;
    }
  }
  if (j >= len2) {
    while (i < len1) {
      data_sort[index] = d1[i];
      i++;
      index++;
    }
  }
}

/*
 * Copy d1 elements from index start with length steps into vector d2
 */
void copyArray(double *d1, double *d2, int start, int length) {
  int i, j = start;
  for (i = 0; i < length; i++) {
    d2[i] = d1[j];
    j++;
  }
}

/*
 * Swaps elem in place i with j in vector
 */
void swap(double *vector, double temp, int i, int j) {
  vector[i] = vector[j];
  vector[j] = temp;
}

/*
 * Checks if the data is sorted
 */
int dbgSorted(double* data, int length) {
  int i;
  for (i = 0; i < length-1; ++i) {
    if (data[i] > data[i+1]) {
      return 0;
    }
  }
  return 1;
}

/*
 * Prints vector double
 */
void print_vector(double* vector, int length) {
  int i;
  for (i = 0; i < length; i++) {
    printf("%f ", vector[i]);
  }
  printf("\n");
}

/*
 * Prints vector int
 */
void print_vector_int(int* vector, int length) {
  int i;
  for (i = 0; i < length; i++) {
    printf("%d ", vector[i]);
  }
  printf("\n");
```

```
}
```

C Makefile

```
#####################################################################
# Makefile for Quick sort (parallell algorithm)
#####################################################################

CC          =   mpicc
CCFLAGS     =   -O3
LIBS        =   -lmpi

main:               main.c
  $(CC) $(CCFLAGS) -o main main.c $(LIBS)

main1:          main1.c
  $(CC) $(CCFLAGS) -o main1 main1.c $(LIBS)
```