Umeå University
Department of Computing Science

Parallel Programming 7.5 p
5DV152

Exercises, Chapter/Topic 5

Submitted 2017-02-03
Author: Lorenz Gerber (dv15lgr@cs.umu.se lozger03@student.umu.se)
Instructor: Lars Karlsson / Mikael Ränner

Contents

Table 1 This table shows the identity values for various operators in C

operator	identity value
&&	1
||	0
&	~ 0
|	0
^	0

1 Introduction

This report is part of the mandatory coursework. It describes the solutions for several chosen exercises from the course book [**?**].

2 Identities for various reduction operators - 5.4

The identity values for various operators are shown in table 1.

3 Rounding errors - 5.5

It was assumed that in this exercise, one does not have to worry about how floats actually are represented in computers with sign, exponent and signigicant field. The interpretation is just based on how many digits are used to represent the given numbers. Hence in a), the sequence of values in the variable sum is $0.0, 2.0, 4.0, 8.0, 1010.0$. The last value is based on the fact that 1008.0 will be rounded from four digits in the register to 1010 with 3 digits in the variable. For b), sum of thread 0 is $0.0, 2.0, 4.0$, and sum for thread 1 is $0.0, 4.0, 1000.0$. After the reduce of sum from both threads, sum will be 1000.

It can be seen that the results from serial and parallel computation differ. This is because for floats, the sequence of addition matters when values are rounded due to too high precision.

4 Default scheduling - 5.6

A program was written to obtain the default scheduling of MP in for loops. Instead of indicating a range, indivdual indices for each thread are given. This is more flexible for various thread/iteration combinations when one thread can process two sequences of indices that don't follow each other. The source code can be found in appendix A.

5 Loop-carried dependence - 5.8

Here the problem of loop-carried dependence can be solved by looking at the actual function or values that result. Instead of using 'back' indices, a more explicit form can be written:

$$(fraci2 + 0.5) \times i \tag{1}$$

6 Thread-safe string tokenizer - 5.16

Here it was asked to provide a string tokenizer that is thread save and does not modify the input string. A wrapper function was written around the thread-save string.h library function strtok_t to provide a version that does not modify the input string. Find below the source code:

```
char *tokenizer(char *str, const char *delim, char **saveptr){
  char *token;

  if(str != NULL){
    char *copied_str = malloc(strlen(str + 1));
    strcpy(copied_str, str);
    token = strtok_r(copied_str,delim, saveptr);
  } else {
    token = strtok_r(NULL, delim, saveptr);
  }

  return token;
}
```

7 Histogram A5.1

Here, it was asked to use OpenMP to make a parallel version of the histogram program in chapter 2. The given serial code from chapter 2 was modified by including `#pragma omp parallel for` above the main 'work' for loop. Below the source code for the `main` function is shown. This seems a good example of how quick and easy some serial program can be made parallel by using OpenMP.

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[]) {
   int bin_count, i, bin;
   float min_meas, max_meas;
   float* bin_maxes;
   int* bin_counts;
   int data_count;
   float* data;
   int thread_count;

   /* Check and get command line args */
   if (argc != 6) Usage(argv[0]);
   Get_args(argv, &bin_count, &min_meas, &max_meas, &data_count);
   thread_count = strtol(argv[5], NULL, 10);

   /* Allocate arrays needed */
   bin_maxes = malloc(bin_count*sizeof(float));
```

```
   bin_counts = malloc(bin_count*sizeof(int));
   data = malloc(data_count*sizeof(float));

   /* Generate the data */
   Gen_data(min_meas, max_meas, data, data_count);

   /* Create bins for storing counts */
   Gen_bins(min_meas, max_meas, bin_maxes, bin_counts, bin_count);

   /* Count number of values in each bin */
#  pragma omp parallel for num_threads(thread_count)
   for (i = 0; i < data_count; i++) {
      bin = Which_bin(data[i], bin_maxes, bin_count, min_meas);
      bin_counts[bin]++;
   }

#  ifdef DEBUG
   printf("bin_counts = ");
   for (i = 0; i < bin_count; i++)
      printf("%d ", bin_counts[i]);
   printf("\n");
#  endif

   /* Print the histogram */
   Print_histo(bin_maxes, bin_counts, bin_count, min_meas);

   free(data);
   free(bin_maxes);
   free(bin_counts);
   return 0;

}  /* main */
```

8 Count sort A5.3

a) the variables `i` and `count` should be made private.

b) No, there are only data dependences left, but by setting private the above mentioned variables, there are no loop-carried dependences left.

c) Several possibilities were considered here. Only moving `memcpy` into the parallel loop does not seem promising, as `a` can not be modified until all threads have finished their work package. Hence, the option would be to set a barrier and put the update of `a` into a parallel for loop. But this feels like a 'step-down' from doing the whole update in one go with the library function `memcpy`. The conclusion is, it does not make sense to parallelize the call to `memcpy`.

d) A parallel version of the 'Count_sort' sort algorithm was implemented using OpenMP. The source code can be found in Appendix B.

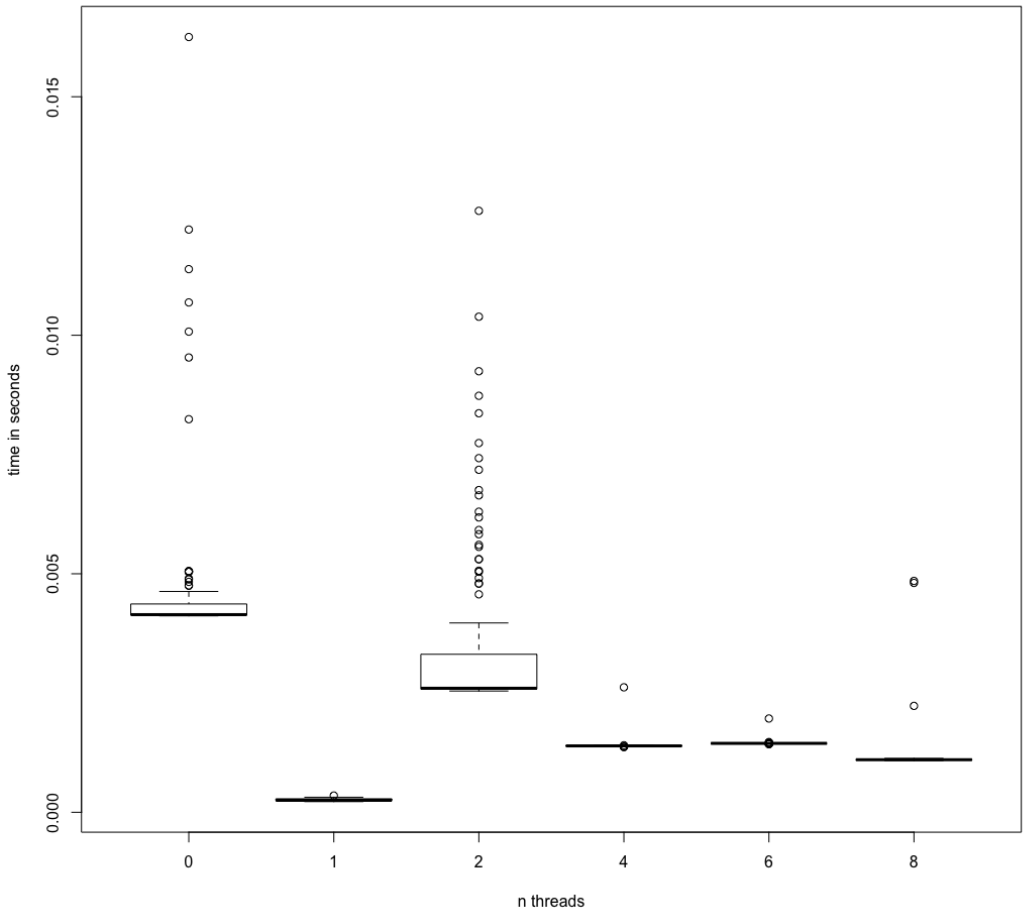

Figure 1: *This boxplot shows the comparison of the parallel OpenMP count sort implementation (n = {2, 4, 6, 8} with the serial countsort (n = 0) and with using the serial quicksort library function (n = 1). It is quite obious that quicksort outperforms also the parallel OpenMP implementation.*

e) For a quick comparison of the performance, a serial implementation with countsort and one with the quicksort library function were compared on a random array with 1000 entries and boxplotted in figure 1. It is obvious that under the tested parameters, 'quicksort' outperforms also the parallel OpenMP implementation. The OpenMP implementation shows compared to the serial algorithm a reasonable speed-up for 2 and 4 threads. Then the curve seems to flatten out. Probably, benchmarking also for larger arrays would establish again better speed-ups. Without having tested for it, I would expect this implementation to be weakly scalable in a reasonable range.

9 Backward substitution A5.4

A C Source Code of Exercise 5.6

```
/**
```

```
 * omp_default.c
 *
 * MP program to determine standard
 * distribution of threads to for
 * loop indices.
 * @usage ./omp_default <threads> <iterations>
 */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char* argv[]){
  int thread_count, iterations, i, j;
  int *assignments;
  int my_rank;

  thread_count = strtol(argv[1], NULL, 10);
  iterations = strtol(argv[2], NULL, 10);

  assignments = (int*) malloc(iterations * sizeof(int));

  # pragma omp parallel for num_threads(thread_count)
  for(i = 0; i < iterations; i++){
    my_rank = omp_get_thread_num();
    assignments[i] = my_rank;
  }

  for(i = 0; i < thread_count; i++){
    printf("Thread %d: Iterations ", i);
    for(j = 0; j < iterations; j++){
      if(assignments[j] == i){
printf("%d ", j);
      }
    }
    printf("\n");
  }

  return 0;
}
```

B C Source Code of Exercise A5.3

```
/**
 * omp_countsort.c
 *
 * OpenMP implementationof count sort algorithm
```

6(7)

```
 * @usage ./omp_countsort <threads> <data_count>
 */
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#include <omp.h>
#include <string.h>


void Count_sort (int a [], int n , int thread_count);

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

  int thread_count, i;
  int *test;
  int data_count = strtol(argv[2], NULL, 10);
  double start, finish;

  test = (int*) malloc(data_count*sizeof(int));

  srand(0);
  for (i = 0; i < data_count; i++)
    test[i] = rand()%101;

  thread_count = strtol(argv[1], NULL, 10);

  for(int i = 0; i < data_count; i++){
    printf("%d ", test[i]);
  }
  printf("\n");

  GET_TIME(start);
  Count_sort(test, data_count, thread_count);
  GET_TIME(finish);

  for(int i = 0; i < data_count; i++){
    printf("%d ", test[i]);
  }

  printf("\n");

  printf("%e\n", (finish - start));

  return 0;
}


void Count_sort (int a [], int n , int thread_count) {
```

```
  int i, j , count;
  //int my_rank = omp_get_thread_num();

  int* temp = malloc ( n *sizeof(int));

# pragma omp parallel for num_threads(thread_count) \
  private(j, count)
  for ( i = 0; i < n ; i ++) {
    count = 0;
    for ( j = 0; j < n ; j ++)
      if ( a[j] < a[i])
count ++;
      else if (a[j] == a[i] && j < i )
count ++;
    temp[count] = a [i];
  }

  memcpy ( a , temp , n *sizeof(int));
  free ( temp );
}
```