Umeå University
Department of Computing Science

Parallel Programming 7.5 p
5DV152

Exercises, Chapter/Topic 5

Submitted	2017-03-02
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 [1].

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 significant 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, individual 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:

$$(\frac{i}{2} + 0.5) \times i \tag{1}$$

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. The source code of the modified `main` function can be found in appendix B. This exercise seems to serve as a good example of how quick and easy some serial programs can be made parallel by using OpenMP.

8 Count sort A5.3

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

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

c) It would be possible to make memcpy parallel by parametrizing the arguments:

```
for (i = 0; i < n; i++) {
    memcpy((a+i), (temp+i), sizeof(int));
}
```

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

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

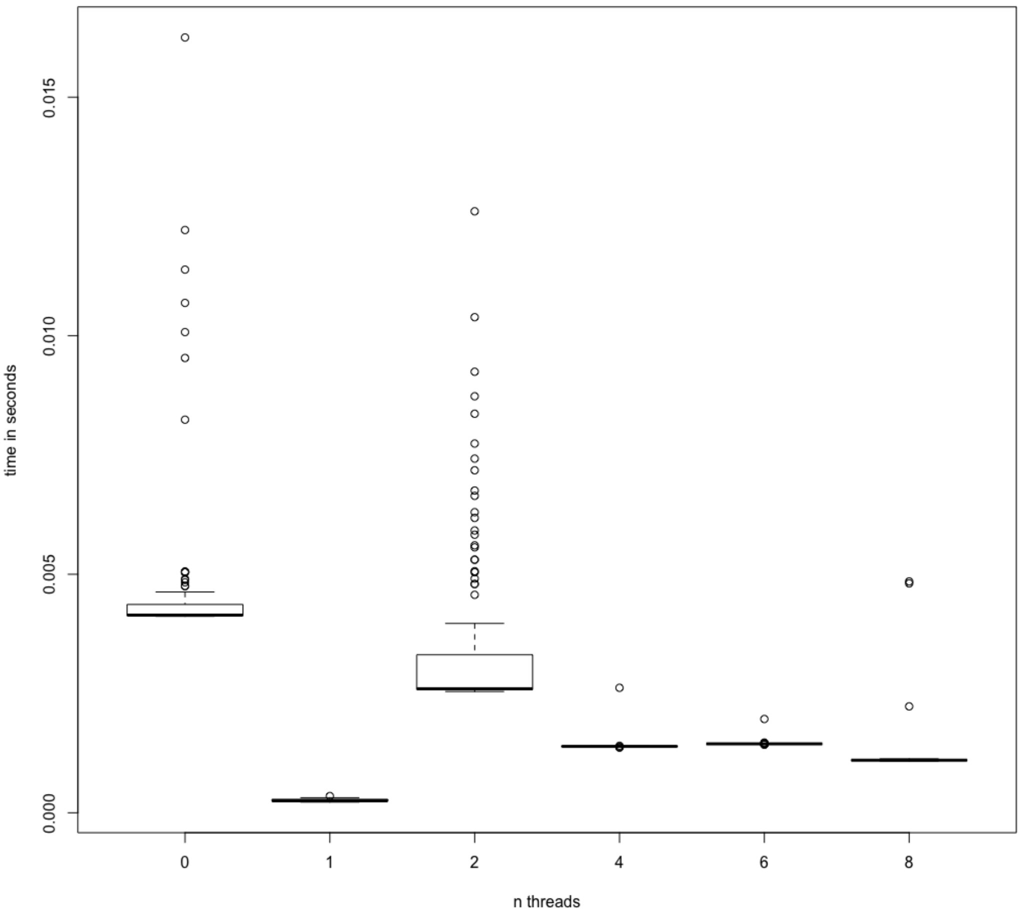

Figure 1: *This boxplot shows the comparison of the parallel OpenMP count sort implementation (n = {2, 4, 6, 8} with the serial countsort (n = 0) and with using the serial quicksort library function (n = 1). It is quite obvious that quicksort outperforms also the parallel OpenMP implementation.*

implementation shows compared to the serial algorithm a reasonable speed-up for 2 and 4 threads. Then the curve flattens out. Benchmarking was run on a 4 core 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) The outer for loop of the row-oriented algorithm can not be parallelized because of the data dependency into the inner loop.

b) The inner for loop of the row-oriented backward substitution is basically a reduce operation onto an array. Hence it can be parallelized. Several attempts were tried. However, apparently, reduce on array is only implemented in OpenMP 4.5 which according to Lars (personal communication) is not supported widely by compilers yet. A work around with several parallel steps should be possible but was not attempted here due to time constraints.

c) The second outer loop of the column-oriented algo can not be parallelized due to the data dependency into the inner loop.

d) The inner loop of the column-oriented can be parallelized. Here a simple `#pragma omp parallel for` directive was used. According to Lars, it should be possible to parallelize the whole algorithm as a block and then use a 'single' directive for the outer loop. Several attempts to obtain this did not work out correct. The suggestion from Lars resulted upon my (I assume incorrect) implementation in segfaults crashes or hanging.

e) The only parallel implementation that worked out was the parallelisation of the inner loop of the column oriented algorithm. The source code can be found in appendix **??**.

f) To evaluate the various scheduling schemes, series of $n = \{1,2,..1024\}$ for the parameter `chunksize` where run in 10 replicates. The default scheduling was used as benchmark. From the data series plotted in figure 2, the best values for chunksize were selected graphically. The lower right subplot of figure 2 shows then the comparison of the best values against the default scheduling, here termed 'auto'. Default scheduling performed best. I assume that in many matrix operations, the block-cyclic scheduling when well adapted to the matrix size will outperform other scheduling types. It would be interesting to know which scheduling is applied here by default and why it is faster than the wide range of the tested alternatives. I assume that just testing for a range is probably also not sufficient as there could be 'jumps' in efficiency from one value to the next if it matches in some wait the underlying data to be scheduled.

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;
```

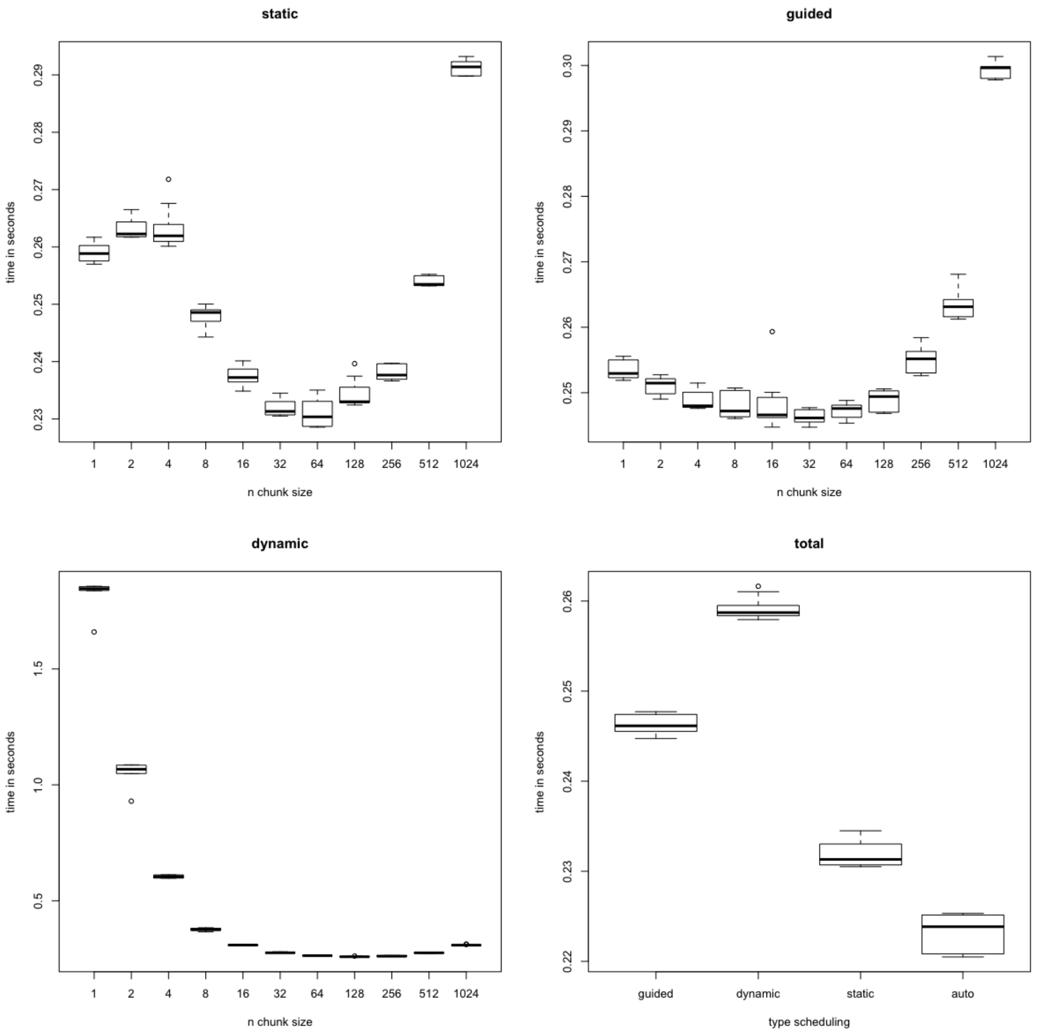

Figure 2: *The four boxplots show a timing (n=10) for different schedulings of the parallel backward substitution algorithm (column bound). The tested schedulings were 'static' (upper-left), 'dynamic' (lower-left) and 'guided' (upper-right), in comparison to 'default' or 'auto' (lower-right).*

```
  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 A5.1

```
#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 */
```

C   C Source Code of Exercise A5.3

```
/**
 * omp_countsort.c
 *
 * OpenMP implementationof count sort algorithm
 * @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];
  }

# pragma omp parallel for num_threads(thread_count)
  for (i = 0; i < n; i++) {
    memcpy((a+i), (temp+i), sizeof(int));
  }


  free ( temp );
}
```

D C Source Code of Exercise A5.4

```
/**
 * omp_backward_substitution.c
 *
 * usage ./backward_substitution <threads> <matrix size n>
 */
#include <stdio.h>
#include <stdlib.h>
#include "timer.h"
#include <omp.h>
#include <string.h>


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

  double** A;
  double* b;
  double* x;
  int thread_count, row, col, i, j;
  int n = strtol(argv[2], NULL, 10);
  double start, finish;

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

  b = malloc(n * sizeof(double));
  x = malloc(n * sizeof(double));
```

```
    // allocate 2d int array size n * n
    A = malloc(n * sizeof(double*));
    for (int i = 0; i < n; i++) {
      A[i] = malloc(n * sizeof(double));
    }

    for (i = 0; i < n; i++){
      for(j = 0; j < n; j++){
        A[i][j] = i+j;
      }
    }

    for (i = 0; i < n; i++){
      b[i] = i;
    }

    GET_TIME(start);

    // backward substitution, column wise
  # pragma for num_threads(thread_count)
    for (row = 0; row < n; row++)
      x[row] = b[row];

    for (col = n-1; col >= 0; col--){
      x[col] /= A[col][col];

  #   pragma omp for num_threads(thread_count)
      for (row = 0; row < col; row++)
        x[row] -= A[row][col]*x[col];

    }

    GET_TIME(finish);

    for(i=0; i< n; i++){
      printf("%f ", x[i]);
    }
    printf("\n");
    printf("%e\n", (finish-start));


    for (i = 0; i < n; i++) {
      free(A[i]);
    }
    free(A);

    return 0;
```

```
}
```

References

[1] P.S. Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufman, 2011.