Parallel Computing

Exercise sheet 2 + Reference Solution

June 3, 2019

Disclaimer

This document contains the assignments from exercise-sheet 2 of the lecture *184.710 Parallel Computing 2019S*, the reference solution as well as my personal solution.

The reference solution is given directly in the assignments in *italics*, my personal solution is always located in the **Solution** subsection of an exercise.

I cannot guarantee the correctness of any solution provided in this document.

Exercise 1

We have seen the three-nested-loop implementation of matrix-matrix multiplication in the lecture. You are now given two $n \times n$ matrices A and B^T, where the matrix B^T is the *transpose* of B. The transposed matrix B is stored in the variable BT. The matrices A, B^T, and C are stored in row-major (C-like) order. We assume that all rows are contiguously stored in memory.

a) *Reference:* $c(n,l) = 2n/l$ *(*n/l *for* A *and* BT*), or precisely* $2\lceil n/l \rceil + 1$

```
for (i=0; i<n; i++) {
    for (j=0; j<n; j++) {
        for (k=0; k<n; k++) {
            C[i][j] += A[i][k] * BT[j][k];
        }
    }
}
```

b) *Reference:* $c(n,l) = n/l + n$ *or* $n + \lceil n/l \rceil + 1$

```
for (i=0; i<n; i++) {
    for (k=0; k<n; k++) {
        for (j=0; j<n; j++) {
            C[i][j] += A[i][k] * BT[j][k];
        }
    }
}
```

c) *Reference:* $c(n,l) = 2n$ *(n for C and A), or* $2n+1$

```
for (j=0; j<n; j++) {
    for (k=0; k<n; k++) {
        for (i=0; i<n; i++) {
            C[i][j] += A[i][k] * BT[j][k];
        }
    }
}
```

1. Examine the three possible variants (a-c) of the triple-nested loop. For each loop variant, give the number of cache misses c over all iterations of the **first execution** of the innermost loop as a function of n and the cache-line size $l, l < n$ (in number of matrix elements that can be stored per line), e.g., $c(n,l) = n + l$. We also assume that data is never evicted from the cache (we count only compulsory misses). Explain your answer in each case.

Solution

1. a)

$$c(n,l) = \frac{2n}{l} + 1$$

This variant has high locality as each iteration accesses only consecutive elements. That means if we partition the input elements into line-sized block we get miss for each block of the $\frac{n}{l}$ blocks the two input array and only one miss for the output-array.

b)

$$c(n,l) = \frac{n}{l} + n + 1$$

This variant has a fixed input-array-value for A (for the first execution) and therefore A produces only one miss. The output-array access consecutive values so it uses locality and produces only $\frac{n}{l}$ misses, while the input array B^T jumps row-wise (n-elements) and because $l < n$ always produces a miss (n misses).

c)

$$c(n,l) = 2n + 1$$

This variant has fixed input-array B^T 2(for the first iteration) so it has 1 miss. The input-array A and the output-array C jump row-wise and therefore both produce n misses.

Exercise 2

Consider this program fragment (a,b are arrays of integers):

```
for (i=0; i<n; i++) {
    int count = 0;
    for (j=0; j<n; j++) {
        if (a[j]<a[i] || (a[j] == a[i] && j < i)) count++;
    }
    b[count] = a[i];
}
for (i=0; i<n; i++) a[i] = b[i];
```

1. What does the program do?

   It sorts array a into array b (by ranking, with elements made different by lexicographic order, as in merge/prefix lecture)

2. What is the asymptotic, sequential work of the code fragment? Give a very short explanation.

   $O\left(n^2\right)$ *operations, two nested loops à* n *iterations*

3. Parallelize the program with OpenMP. Modify the program by inserting pragmas.

   Reference:

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    int count = 0;
    for (j=0; j<n; j++) {
        if (a[j]<a[i] || (a[j] == a[i] && j < i)) count++;
    }
    b[count] = a[i];
}
#pragma omp parallel for
for (i=0; i<n; i++) {
    a[i] = b[i];
}
```

4. Given p threads, what is the asymptotic, parallel running time of your parallelization as a function of n and p (assuming $n \geq p$)?

   running time: $O\left(\frac{n}{p}\right)$, *or* $O\left(\frac{n^2}{p} + n\right)$ *(if* $n < p$*)*

5. What is the relative and absolute speed-up of your implementation? For the absolute speedup, use the best known asymptotic running time for this type of algorithm. (We also assume that the integers stored in array `a` are much larger than a reasonable array size.)

   relative speed-up: $\frac{n^2}{\frac{n^2}{p}} \in O(p)$ *for* $p \leq n$, *absolute speed-up:* $\frac{n \log n}{\frac{n}{p} n} = \frac{p \log n}{n}$

Solution

1. The algorithm sorts the input-array `a` and uses `b` as temporary buffer.

2. The algorithm works with $O\left(n^2\right)$ instructions because it takes each input element (n times) and compares it with each input element. The copy of the temporary array to the input-array can asymptotically dismissed.

3.

```
#pragma omp parallel for
for (i=0; i<n; i++) {
    int count = 0;
    for (j=0; j<n; j++) {
        if (a[j]<a[i] || (a[j] == a[i] && j < i)) count++;
    }
    b[count] = a[i];
}
#pragma omp parallel for
for (i=0; i<n; i++) a[i] = b[i];
```

4. The parallelization works in $T_{par}(p,n) = O\left(\frac{n^2}{p}\right)$

5. The relative speed-up is:

$$\begin{aligned} SRel_p(n) &= \frac{T_{par}(1,n)}{T_{par}(p,n)} \\ &= \frac{n^2}{\frac{n^2}{p}} \\ &= p \end{aligned}$$

The absolute speed-up is:

$$\begin{aligned} S_p(n) &= \frac{T_{seq}(n)}{T_{par}(p,n)} \\ &= \frac{n \log n}{\frac{n^2}{p}} \\ &= \frac{p \log n}{n} \end{aligned}$$

Exercise 3

Consider the following C code, which has been parallelized with OpenMP:

```c
int i;
int j = 0;
int result[n];

#pragma omp parallel for
for (i=0; i<n; i++)  {
    int k;
    if (predicate(i)) {
#pragma omp atomic capture
        k = j++;

        result[k] = i;
    }
}
```

The code snippet is supposed to record in the `result` array the indices for which the predicate holds.

1. Does the following condition hold after the loop: `result` $[k] <=$ `result` $[k+1]$ for all k, $0 \leq k < n$? Justify if true or not true.

   No; the thread order in which the atomic fetch and increment is executed is not determined. The k's computed by the threads are successive, but this is not so for the loop indices written into the result array.

2. What is the asymptotic work of the program as a function of n, assuming that the predicate evaluation and the atomic operation both take $O(1)$ steps?

   work: $O(n)$

3. What is the parallel time with p threads, p (much) smaller than n? You can assume that the atomic operation take $O(1)$ time, independent of the number of threads.

   time: $O(n/p)$

4. How many indices are stored in the result array?

   Value of j after the loop. This is the number of indices for which the predicate holds.

Solution

1. This is not true because the way how the `for`-loop is split up, doesn't has to be sequential.

2. The work is $W(n) = O(n)$.

3. $T_{par}(p, n) = O\left(\frac{n}{p}\right)$

4. The number of indices for which the predicate returns `true`.

Exercise 4

Consider the following OpenMP code snippet that is supposed to count the number of times some event happened.

```
int event[p];
#pragma omp parallel
{
    int t = omp_get_thread_num();

    event[t] = 0;
    while (running) {
        if (eventhappened()) {
            event[t]++;
        }
        // ...
    }
}
```

Assume that the number of iterations is large and balanced across the threads (no load balancing issue), that events are frequent, and that the other computations done in the loop are fast.

1. What is a potential performance problem with this way of counting events? Explain. You can also assume that `eventhappened()` is thread-safe and takes time $O(1)$.

   Performance-problem: False sharing as the counter array resides in one cache line.

2. Propose a better solution. Explain the changes that you would do in order to solve the potential problem. Modify the code accordingly.

   Padding or local variable per thread.

Solution

1. The problem is false sharing. The event counts are updated independently but they share the same cache line and therefore needs to maintain cache coherence.

2.

```
int event[p];
#pragma omp parallel
{
    int t = omp_get_thread_num();

    int count = 0;

    while (running) {
        if (eventhappened()) {
            count++;
        }
        // ...
    }
    event[t] = count;
}
```

Exercise 5

You are given the following, manual parallelization of a loop, which should assign `a[i] = i` for all values of `i` between `0` and `n-1`.

```
int p, n1, n2;

#pragma omp parallel
{
    int p = omp_get_num_threads();
    int t = omp_get_thread_num();
    int i;

    n1 = t*(n/p);
    if (t<n%p)
        n1 += t;
    else
        n1 += n%p;

    n2 = n1+n/p;
    if (t<n%p)
        n2++;

    for (i=n1; i<n2; i++) {
        a[i] = i;
    }
}

printf("Threads␣%d\n", p);
```

1. What are the problems with this parallelization? Is it a correct OpenMP parallelization? Find three problems and explain the problems.

   Race condition, unguarded updates to `n1, n2` (counts as 2 problems!), global variable `p` is undefined after the parallel region.

2. What is an immediate remedy, e.g. by using OpenMP features?

   Possible solution:

```
int p, n1, n2;

#pragma omp parallel private(n1,n2) shared(p)
{
#pragma omp single
      p = omp_get_num_threads();
  int t = omp_get_thread_num();
  int i;

  n1 = t*(n/p);
  if (t<n%p) n1 += t;
  else n1 += n%p;

  n2 = n1+n/p;
  if (t<n%p) n2++;

  for (i=n1; i<n2; i++) {
      a[i] = i;
  }
}

printf("Threads %d\n", p);
```

Solution

1. a) `n1` and `n2` need to be thread private. Because they depend on the `thread_num` and are written in every thread and therefore race condition occurs.

   b) The variable `p` is declared outside of the parallel block and is never assigned a value because it is declared again (and therefore hidden) in the parallel block. As a result the `printf` statement tries to access an uninitialized value.

   c)

2. a) For `n1, n2` the solution is to extend the `pragma` by a private-clause:

   `#pragma omp parallel private(n1,n2)`

   b) The solution for `p` is to remove the declaration outside of the parallel block and move the printf statement inside the parallel block guarded by

   `#pragma omp single nowait`

Exercise 6

The following program gets a matrix x of size $n \times n$ as an input and should parallelize the n row-wise additions into the first element of each row (appeared in lecture).

```
for (i=0; i<n; i++) {
#pragma omp parallel for
    for (j=1; j<n; j++) {
        x[i][0] += x[i][j];
    }
}
```

1. Why is the program incorrect? Explain.

   Classical race condition, outcome in `x[i][0]` undefined.

2. How can it made correct by exploiting OpenMP functionality? You are not allowed to move the pragma. Present the updated code.

   Use parallel reduction clause

```
for (i=0; i<n; i++) {
    v = &x[i][0];
#pragma omp parallel for reduction(+:v[0])
    for (j=1; j<n; j++) {
        v[0] += x[i][j];
    }
}
```

3. What is the asymptotic running time of your updated, corrected code (as a function of n and p, where p is the number of threads)?

   running time $O\left(n^2/p\right)$

Solution

1. The program contains a race condition on the element `x[i][0]`.
2. By using the openMP reduction Feature:

   `#pragma omp parallel private(j) reduction(+:x[i][0])`
3. The asymptotic running time is $T_{par}(p, n) = O\left(\frac{n^2 - n}{p}\right)$

Exercise 7

You are given the following code snippet. The `sleep_s` function takes a value `w[i]` as input and waits as long as specified by `w[i]`.

```
int w[] = {1, 10, 3, 1, 5, 12, 2, 7, 1, 6, 2};
int n = sizeof(w)/sizeof(int);

#pragma omp parallel for schedule(runtime)
for (i=0; i<n; i++) {
    sleep_s(w[i]);
}
```

1. We assume `OMP_NUM_THREADS=4` (thus $p = 4$). For example, iteration 0 has weight 1 and therefore the length of the loop iteration is 1s. Draw Gantt charts of the resulting schedules for each of the three scheduling policies listed below. If multiple threads are ready at the same time, assign the loop iteration(s) to the thread with the smaller thread identifier. We assume, that threads are identified ba a unique number $t, 0 \leq t < p$.

   a) static

   b) static(4)

   c) dynamic

   Note that for `static` we make following assumptions. The OpenMP 5.0 standard (page 105) states: „For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p \cdot q - r$, with $0 \leq r < p$. One compliant implementation of the static schedule (with no specified `chunk_size`) would behave as though `chunk_size` had been specified with value q."In this exercise, you need to use this compliant implementation specification given above when drawing the Gantt chart for `static`.

   Reference solutions are the same diagrams as in the personal solution.

Solution

1. a) `static`: Gantt chart in figure 1 on the following page

   b) `static(4)`: Gantt chart in figure 2 on the next page

   c) `dynamic`: Gantt chart in figure 3 on the following page

Exercise 8

Consider the following parallel code, which computes each pixel of a Julia set (taken and adapted from `https://people.sc.fsu.edu/~jburkardt/c_src/julia_openmp/julia_`

Figure 1: Gantt-chart for `static` scheduling.

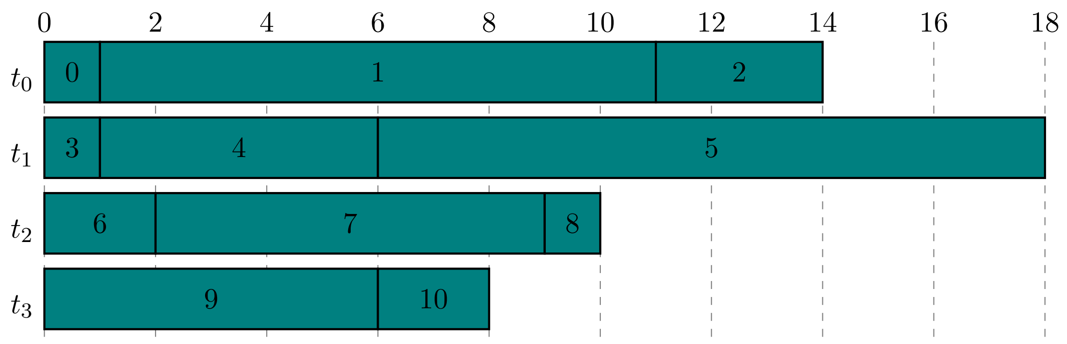


Figure 2: Gantt-chart for `static(4)` scheduling.

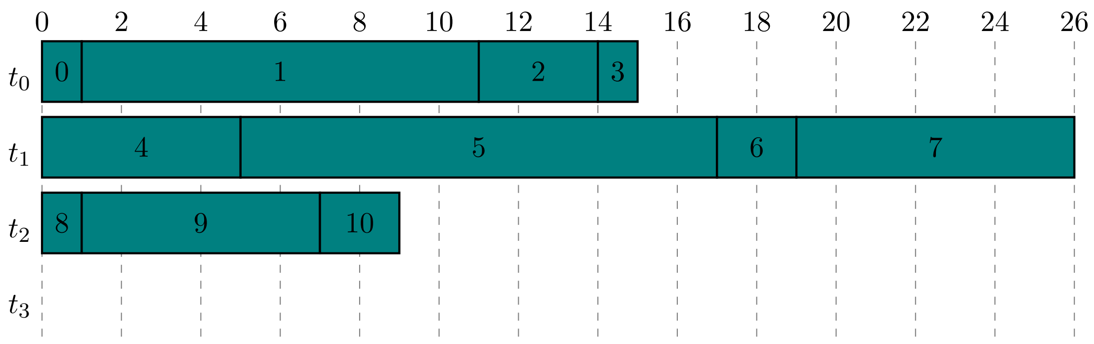
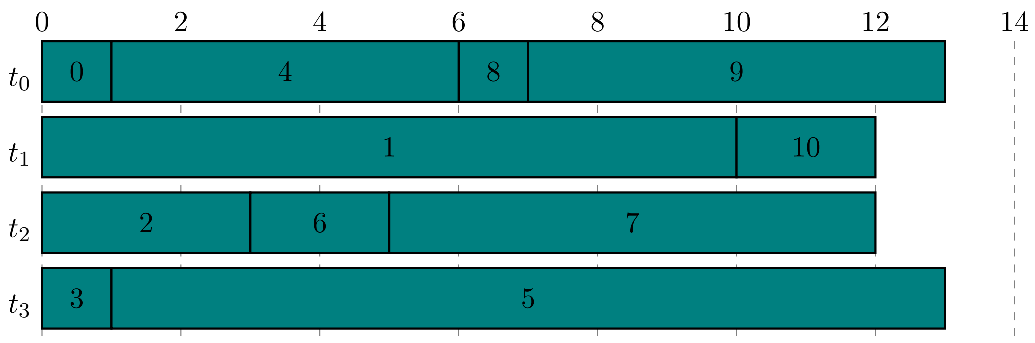


Figure 3: Gantt-chart for `dynamic` scheduling.



openmp.html). For each pixel in the height (h) and width (w) dimension of an RGB image, the code computes one juliaValue. The computation of this value (line 13) is highly heterogeneous in term of running-time, as a maximum of 200 iterations in a computational loop are performed (in case of a red pixel). However, if in some iteration a given threshold is exceeded, then the corresponding pixel does not belong to the Julia set and the loop is terminated earlier. Thus, a line containing only white pixels should have a smaller running time than a line containing many red pixels.

```
1  rgb = (unsigned char *) malloc(w * h * 3 * sizeof(unsigned char));
2  
3  # pragma omp parallel \
4    shared (h, w, xl, xr, yb, yt) \
5    private (i, j, k, juliaValue)
6    {
7  # pragma omp for collapse(2) schedule(runtime)
8      for (j=0; j<h; j++) {
9        for (i=0; j<h; j++) {
10         juliaValue = julia(w, h, xl, xr, yb, yt, i, j);
11 
12         k = 3 * (j * w + i);
13 
14         rgb[k] = 255 * (1 - juliaValue);
15         rgb[k + 1] = 255 * (1 - juliaValue);
16         rgb[k + 2] = 255;
17       }
18     }
19    }
```

1. We ran this Julia application on one compute node of hydra (which has a maximum of 32 cores) to generate a Julia image of size 10 000 × 10 000, as shown in figure 4 on page 16. We used three different configurations to obtain the same image of a Julia set. When we executed the program on a compute node and used only 1 core (1 thread), the program completed its execution in 35 s. Now, we set OMP_NUM_THREADS=32 and start with either:

   a) OMP_SCHEDULE="static"

   b) OMP_SCHEDULE="dynamic,100"

   c) OMP_SCHEDULE="dynamic,1"

   We have measured the following running times for the latter configurations: 6.9s, 2.73s, 1.04s. Assign each running time to one of the configurations and explain your reasoning (hint: scheduler overhead, workload distribution).

   dynamic,100 = 1.04 s, dynamic,100 has a good granularity and dynamic can balance the heterogeneous workload.

static = 2.73 s, static has almost no scheduler overhead but suffers from uneven workload distributions.

dynamic,1 = 6.9 s, dynamic,1 is too fine grained and the scheduler is the bottleneck.

Note: Was graded according to reasoning NOT the correct order (as the order can change per system (hydra vs. PC) and even in-between runs)!

Solution

1. a) `OMP_SCHEDULE="static"`: 6.9 s

   Because each pixel is allocated to a thread beforehand and the computation time is highly heterogeneous a thread can have „bad luck " and receive many high load pixels which results in severe load in-balance.

   b) `OMP_SCHEDULE="dynamic,100"`: 1.04 s

   Because the chunk size is much less than the overall values (100 << 100 000 000) the scheduling is very dynamic, meaning that a finished thread gets new work instead of idling, but has less overhead than a dynamic scheduling with chunk-size 1.

   c) `OMP_SCHEDULE="dynamic,1"`: 2.73 s

   The dynamic scheduling is much faster than static scheduling because while a thread is occupied with a high workload the other threads can calculate multiple less intensive calculation, but they need to fetch a new value very often which results in a higher scheduler-overhead compared to dynamic scheduling with a greater chunk size.

Exercise 9

You are given the following OpenMP program:

```
int max = 0;
int col_start, col_end;

#pragma omp parallel
{
    int i, j;
    int p = omp_get_num_threads();
    int tid = omp_get_thread_num();
    col_start = tid * (n/p);
    col_end = (tid+1) * (n/p);
    if (tid == p - 1)
        col_end += n%p;

    for (i=0; i<n; i++) {
```

```
15          for (j=col_start; j<col_end; j++) {
16  #pragma omp critical
17              if (max < a[i][j])
18                  max = a[i][j];
19          }
20      }
21      // we are done
22      printf("maximum␣is:␣%d\n", max);
23  }
```

The program should compute the maximum values of all cells in the matrix a with $n \times n$ elements. All numbers in a are strictly greater than 0. The code contains three problems, which are either correctness or performance problems. (That the remainder is given to the last thread is not one of those problems. For large n, this problem is irrelevant.)

1. Identify the three problems and explain why these lines are problematic.

   race on `col_start` and `col_end`, race on `max` in `printf` (needs barrier), critical section called too often (performance problem)

2. Present the updated code that fixes the problems.

   No reference

Solution

1. a) `col_start` and `col_end` in line 2 are shared in the parallel region and therefore have a race condition and need to be thread private.

   b) The critical region in line 16 poses a performance problem, because all threats have to wait even if the maximum has not been found.

   c)

2.

```
1      int max = 0;
2      int col_start, col_end;
3
4      #pragma omp parallel private(col_start, col_end) reduction(max:max)
5      {
6      int i, j;
7      int p = omp_get_num_threads();
8      int tid = omp_get_thread_num();
9      col_start = tid * (n/p);
10     col_end = (tid+1) * (n/p);
11     if (tid == p - 1)
12         col_end += n%p;
13
```

```
    for (i=0; i<n; i++) {
        for (j=col_start; j<col_end; j++) {
            if (max < a[i][j])
            max = a[i][j];
        }
    }
    // we are done
    printf("maximum␣is:␣%d\n", max);
}
```

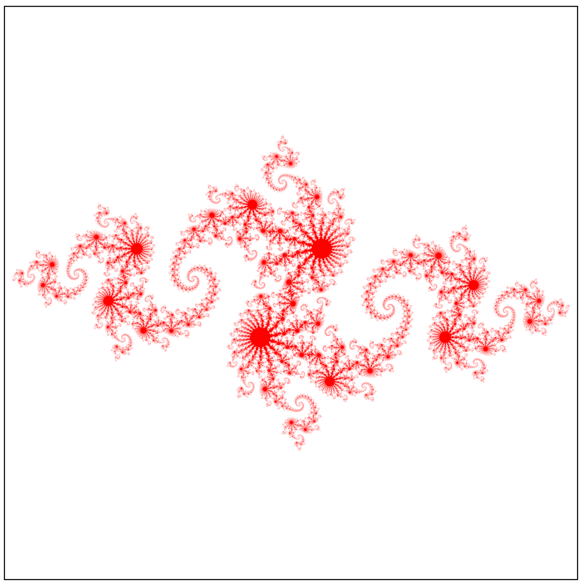


Figure 4: Julia set of size 10 000 × 10 000 pixels. The white area within the bounding box belongs to the computed image.