Assoc.Prof. Dr. Sascha Hunold
TU Wien
Faculty of Informatics
Research Group for Parallel Computing

Basics of Parallel Computing
2021S
Assignment 2

issue date: 2021-05-19
due date: 2021-06-02
No extensions.

WHAT TO HAND IN VIA TUWEL

1. A PDF file with the solutions to all exercises and the plots.
2. The source code of your implementations. For this assignment, you will have to edit the following files from `julia-student-1.0.0-Source.tar.bz2`:

   ```
   juliap.c filter.c
   ```

   After you have implemented these functions, upload your modified files

   a) `juliap.c` and

   b) `filter.c`

   to TUWEL. Double check that you do not upload the original files!

Exercise 1 (4 [1+2+1] points)

In the following loop, which is parallelized with OpenMP, the concrete schedule is decided at runtime by setting the `OMP_SCHEDULE` environment variable.

```
1 #pragma omp parallel for schedule(runtime)
2 for (i=0; i<n; i++) {
3   a[i] = omp_get_thread_num();
4   t[omp_get_thread_num()]++;
5 }
```

The loop is run with 4 threads (by setting `OMP_NUM_THREADS=4`) and with `n=15` iterations.

1. What do `a` and `t` count?
2. Give the values for all elements in `a` and `t` with
   - `OMP_SCHEDULE="static"`
   - `OMP_SCHEDULE="static,1"`
   - `OMP_SCHEDULE="static,3"`

   Show possible values (one possibility is enough) for all elements in `a` and `t` with
   - `OMP_SCHEDULE="dynamic,1"`
   - `OMP_SCHEDULE="dynamic,5"`
   - `OMP_SCHEDULE="guided,2"`

   To answer this question, fill the following tables. The individual table cells should contain the values found in either array, `a` and `t`, respectively.

case / *a*	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`static`		insert thread ids													
`static,1`															
`static,3`															
`dynamic,1`															
`dynamic,5`															
`guided,2`															

case / *t*	0	1	2	3
`static`				
`static,1`				
`static,3`				
`dynamic,1`				
`dynamic,5`				
`guided,2`				

3. What is a possible performance problem with the assignment to the `t` array (Line 4)?

Exercise 2 (3 points)

A developer had the task to parallelize the function `omp_odd_counter`, which computes the number of all odd numbers in an array `a` with `n` elements. The first attempt of the programmer is shown in Listing 1.

Listing 1: Parallelized code.

```
int omp_odd_counter(int *a, int n)
{
  int i;
  int count_odd = 0;

  #pragma omp parallel for shared(a) private(count_odd)
  for(i=0; i<n; i++) {
    if( a[i] % 2 == 1 ) {
      count_odd++;
    }
  }

  return count_odd;
}
```

1. Fix the problems with this OpenMP code, such that the computation will always be correct. Give a brief explanation for the required fixes.

Exercise 3 (4 points)

We have the following three OpenMP programs:

Listing 2: Version A.

```
int omp_tasks(int v)
{
  int a, b;
  if( v <= 2 ) {
    return 1;
  } else {
    #pragma omp task shared(a,b)
    a = omp_tasks(v-1);
    #pragma omp task shared(a,b)
    b = omp_tasks(v-2);
    #pragma omp taskwait
    return a + b;
  }
}

void main() {
  int res;
  #pragma omp parallel
  {
    #pragma omp master
    res = omp_tasks(5);
  }
  printf("res=%d\n", res);
}
```

Listing 3: Version B.

```
int omp_tasks(int v)
{
  int a, b;
  if( v <= 2 ) {
    return 1;
  } else {
    #pragma omp task shared(a,b)
    a = omp_tasks(v-1);
    #pragma omp task shared(a,b)
    b = omp_tasks(v-2);
    #pragma omp taskwait
    return a + b;
  }
}

void main() {
  int res;
  #pragma omp parallel
  {
    res = omp_tasks(5);
  }
  printf("res=%d\n", res);
}
```

Listing 4: Version C.

```
int omp_tasks(int v)
{
  int a, b;
  if( v <= 2 ) {
    return 1;
  } else {
    #pragma omp task shared(a,b)
    a = omp_tasks(v-1);
    #pragma omp task shared(a,b)
    b = omp_tasks(v-2);
    #pragma omp taskwait
    return a + b;
  }
}

void main() {
  int res;
  #pragma omp master
  res = omp_tasks(5);
  printf("res=%d\n", res);
}
```

1. What is the output of the three different versions of the program when the programs are executed with `OMP_NUM_THREADS=4`?
2. How often is the function `omp_tasks(int v)` called in each version of the program when being executed with `OMP_NUM_THREADS=4`? Explain your reasoning!

Exercise 4 (10 [2 (code) + 4 (strong scaling) + 4 (schedule)] points)

We return to the code to generate an image of a Julia set from the last assignment. We have ported the Python code to C. The actual C code to compute Julia set is located in the function `compute_julia_set` in file `juliap.c`.

To compile the program, do the following after unpacking the tarball:

```
cmake .
make
```

You can now start the program like this:

```
./bin/juliap_runner -n 10 -p 2
```

If you want to check whether the program is working correctly, you can convert the Julia set into an image like this:

```
# write output to test.out
./bin/juliap_runner -n 1000 -p 2 -o test.out

# convert test.out into png file
python ./contrib/julia2img.py -i test.out -o test.png
```

The output file should look like the one shown in Figure 1.

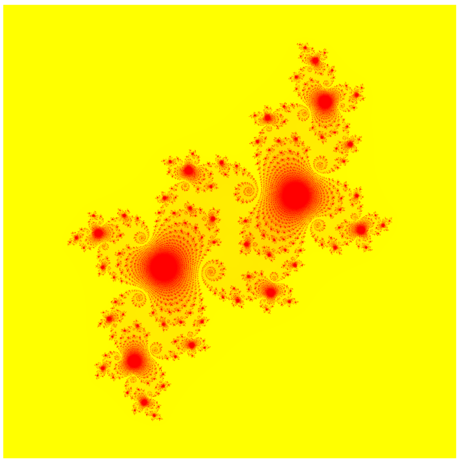

Figure 1: Example Julia image.

Your task is now to parallelize the computation of each pixel using OpenMP. To do so, you need to apply the `for` directive to the two nested loops. Check whether you can apply the `collapse` clause to increase the potential parallelism. In any case, append the `schedule(runtime)` clause to the line containing the `for` directive. By applying `schedule(runtime)`, you can select the loop scheduling strategy by setting an environment variable (compare to Exercise 1).

1. Measure the running time of your parallelized code for the following cases on one of the compute nodes of hydra:
   - $n \in \{100, 1000\}$ and
   - $p \in \{1, 2, 4, 8, 16, 24, 32\}$.

   Repeat each of these experiments 3 times! For these experiments, do not set the `OMP_SCHEDULE` variable.
   - Plot the minimum running time obtained from the three experiments (for each case) for both input sizes of n, i.e., 1 plot for $n = 100$, x-axis: number of cores, y-axis: running time in seconds, and another plot for $n = 1000$.
   - Discuss your findings.

2. Now, we analyze how the schedule parameter influences the running time. To that end, measure the running time of your parallelized code for the following cases on one of the compute nodes of hydra:
   - $n \in \{1000\}$,
   - $p \in \{16\}$, and
   - `OMP_SCHEDULE` $\in$ { `static` | `static,1` | `dynamic,1` | `guided,10` }
     ("`|`" is used to separate the different elements to avoid ambiguity.).

   As always, repeat each experiment 3 times.
   - Plot the minimum running time obtained from the three experiments (for each case) for the different schedule options, i.e., 1 plot for $n = 1000$, x-axis: schedule option, y-axis: running time in seconds.
   - Discuss your findings. Try to explain why we see or we do not see performance differences.

Exercise 5 (12 [4 (code) + 4 (strong scaling) + 4 (weak scaling)] points)

The last exercise is concerned with applying a Gaussian filter to a PNG image. We are going to use the images produced by the previous exercise. To that end, we apply a convolution filter to each pixel of the PNG image. For an introduction to convolution, see `https://en.wikipedia.org/wiki/Kernel_(image_processing)`, which also contains an example of the Gaussian blur filter that we are going to apply to our images. However, we will compute the convolution in parallel, i.e., we will use OpenMP to parallelize our code.

The convolution code is already given in the function `apply_filter` in file `filter.c`. Your task is now to parallelize the function `apply_filter` using the `task` directive of OpenMP. Think about the right granularity when spawning a new OpenMP task. Note: Although you could potentially use the `for` directive, the `for` directive is not allowed to complete this task.

The binary `filter_runner` is already built after typing

```
cmake .
make
```

in the previous exercise. You can start the filter program like this:

```
./bin/filter_runner -i ./contrib/input1.png -p 1
```

If you want to see the output of the filter, you can call

```
./bin/filter_runner -i ./contrib/input1.png -p 1 -o foo.png
```

The file `foo.png` will then contain the resulting image.

1. Measure the running time of your parallelized code for the following cases on one of the compute nodes of hydra:
   - $i \in \{$`input1.png`$\}$,
   - $p \in \{1, 2, 4, 8, 16, 24, 32\}$, and
   - $r \in \{1\}$.

   Repeat each of these experiments 3 times.
   - Plot the minimum running time obtained from the three experiments (for each case), i.e., 1 plot, x-axis: number of cores, y-axis: running time in seconds.
   - Discuss your findings.

2. Measure the running time of your parallelized code for the following cases on one of the compute nodes of hydra:
   - $i \in \{$`input1.png`$\}$,
   - $p \in \{1, 2, 4, 8, 16, 24, 32\}$, and
   - $r \in \{1, 2, 4, 8, 16, 24, 32\}$.

   Here, we do a weak scaling experiment. So, when we run with $p = 1$, we also select $r = 1$ and with $p = 2$ then $r = 2$ and so on. **Repeat** each of these experiments 3 times.
   - Plot the minimum running time obtained from the three experiments (for each case), i.e., 1 plot, x-axis: number of cores, y-axis: running time in seconds.
   - Discuss your findings.

A. Appendix

A.1. Unpacking the tarball and compiling the code

Once you have logged into `hydra`, you can unpack julia-student-1.0.0-Source.tar.bz2. We assume that you have created a directory called `project` and julia-student-1.0.0-Source.tar.bz2 is located inside this directory.

You can now unpack the tarball and compile the code:

```
stester@hydra:~/project$ tar xfvj julia-student-1.0.0-Source.tar.bz2
stester@hydra:~/project$ cd julia-student-1.0.0-Source/
stester@hydra:~/project/julia-student-1.0.0-Source$ cmake .
stester@hydra:~/project/julia-student-1.0.0-Source$ make
```

Once this is done, you can test your code on the front-end machine (`hydra`). Note: For the actual runtime experiments, you must use one of the compute nodes of `hydra` (see Section A.2).

```
stester@hydra:~/project/julia-student-1.0.0-Source$
  ./bin/juliap_runner -n 100 -p 1
100,1,0.158502
```

A.2. Running the Experiments / Using SBATCH Files

We also provide SBATCH files that you can use to run your experiments. You can find them in the `jobs_files` directory inside the tarball. Please modify the variables in the script according to your needs and the actual experiment. You can then simply submit the sbatch file to SLURM:

```
sbatch run_juliap.job
```

A.3. Running the Experiments / Using `srun`

You could also run your experiments in a more interactive way by using `srun`. When you call `srun` on the front-end machine `hydra`, you will get an allocation on one of the available compute nodes. On this free compute node, your command will be executed.

```
stester@hydra:~/project/julia-student-1.0.0-Source$
  srun -N 1 -p q_student -t 2 ./bin/juliap_runner -n 1000 -p 1
1000,1,16.068483
```

You could complete your measurment tasks entirely with `srun`. We still recommend to use the provided `sbatch` file.

A.4. Optional: Docker

Important: Docker containers can help you developing and testing the code on your private machine. The running time experiments still have to be performed on `hydra`.

You can use Docker to build a container that provides all required build tools and a Python interpreter. A `Dockerfile` is part of julia-student-1.0.0-Source.tar.bz2. We provide this `Dockerfile` without any warranty that it works in your environment, but it may be helpful to some.

You can run the following command to build the container (assuming you are in the directory where the `Dockerfile` resides):

```
docker build -t bopc2 .
```

When the build has been completed, you can start the container like this

```
docker run -it --rm -v "$PWD":/home/ bopc2  bash
```

(`$PWD` is your current working directory. You may need to change this variable depending on your operating system.)

Once you have started the container, you can go to the `/home` directory and compile your code like this:

```
root@b6182f39fa34:/home# cmake .
-- The C compiler identification is GNU 10.3.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Found OpenMP_C: -fopenmp (found version "4.5")
-- Found OpenMP: TRUE (found version "4.5")
-- Found ZLIB: /usr/lib/x86_64-linux-gnu/libz.so (found version "1.2.11")
-- Found PNG: /usr/lib/x86_64-linux-gnu/libpng.so (found version "1.6.37")
-- Configuring done
-- Generating done
-- Build files have been written to: /home

root@b6182f39fa34:/home# make
Scanning dependencies of target filter_runner
[ 57%] Built target filter_runner
Scanning dependencies of target juliap_runner
[100%] Built target juliap_runner

root@b6182f39fa34:/home# ./bin/juliap_runner -n 100 -p 1
100,1,0.119430
```