TU WIEN Informatics

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

Basics of Parallel Computing
2025S
Assignment 2

---

issue date: 2025-04-15
due date: 2025-05-30

Please do not forget to register with a group on TUWEL!
(Otherwise, you will not be able to submit your solution.)

WHAT TO HAND IN VIA TUWEL

1. A PDF file with the solutions to all exercises and the plots.
2. Option 1: 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!
3. Option 2: **After** you have completed the GPU code, upload
   - `kernel_gpu.cu`

   to TUWEL.

Rules

- Ensure that the following information is provided in the PDF document:
  - names of authors,
  - date, and
  - course and assignment name.
- Use a spell checker. Check your grammar.
- Label your figures and tables appropriately, e.g., "Table 1: Summary" is not an appropriate description.
- Reference each figure and table in the text. Ensure that each figure/table is indeed described in the text. Only stating that "Table 1 contains a summary." is not considered appropriate. Please explain what exactly is summarized in a table.
- Provide your raw benchmark data in the appendix of your submission report. When using LaTeX, you may want to have a look at the csvsimple package to generate tables from your csv files.

Exercise 1 (3 [1+2] 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 3 threads (by setting `OMP_NUM_THREADS=3`) 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,2"`
   - `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,4"`
   - `OMP_SCHEDULE="guided,7"`

   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,2`															
`static,3`															
`dynamic,1`															
`dynamic,4`															
`guided,7`															

case / *t*	0	1	2
`static`			
`static,2`			
`static,3`			
`dynamic,1`			
`dynamic,4`			
`guided,7`			

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

Let us assume that we know the number of iterations n and the time needed (in time units) for each iteration, given by `time[i]`, where $0 \leq i < n$. In addition, we assume that there is no overhead associated with loop scheduling, i.e., fetching a new block in case of dynamic schedules has zero cost (0 time units).

The following workload is given:

`i`	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
`time[i]`	1	2	3	1	1	1	3	2	3	2	1	2	10	2	4	3	3	12

1. Given $p = 4$ cores, what would be the length of the shortest possible schedule (the minimum makespan)? Note that, in this case, the iterations may be executed in any order. Present the Gantt chart of this best schedule.

2. Show the Gantt chart with `static,3`. What is the resulting makespan?

3. Show a possible Gantt chart with `dynamic,2`. What is the resulting makespan?

Exercise 3 (4 points)

We have the following three OpenMP programs:

Listing 1: Version A.

```
int omp_tasks(int v)
{
  int a = 0, b = 0;
  if( v <= 2 ) {
    return 1;
  } else {
    #pragma omp task shared(a)
    a = omp_tasks(v-1);
    #pragma omp task private(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 2: Version B.

```
int omp_tasks(int v)
{
  int a = 0, b = 0;
  if( v <= 2 ) {
    return 1;
  } else {
    #pragma omp task shared(a)
    a = omp_tasks(v-1);
    #pragma omp task shared(b)
    b = omp_tasks(v-2);
    #pragma omp taskwait
    return a + b;
  }
}

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

Listing 3: Version C.

```
int omp_tasks(int v)
{
  int a = 0, b = 0;
  if( v <= 2 ) {
    return 1;
  } else {
    #pragma omp task shared(a)
    a = omp_tasks(v-1);
    #pragma omp task shared(b)
    b = omp_tasks(v-2);
    #pragma omp taskwait
    return a + b;
  }
}

void main() {
  int res;
  #pragma omp parallel
  {
    #pragma omp critical
    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=3`?

2. How often is the function `omp_tasks(int v)` called in each version of the program when being executed with `OMP_NUM_THREADS=3`? Explain your reasoning!

Option 1

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
python3 ./contrib/julia2img.py -i test.out -o test.png
```

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

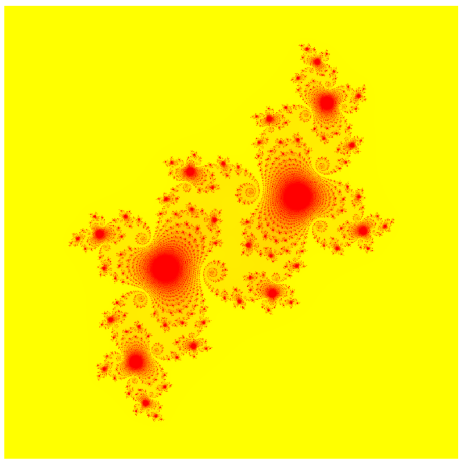

Figure 1: Example Julia image.

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. Parallelize the pixel computation in `juliap.c` using OpenMP `for` directives.

2. Measure the running time of your parallelized code for the following cases on one of the compute nodes of hydra:

   - $n \in \{85, 1150\}$ and
   - $p \in \{1, 2, 4, 8, 16, 24, 32\}$.

   Repeat each of these experiments 3 times! In 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 $n = 85$, x-axis: number of cores, y-axis: running time in seconds, and another plot for $n = 1150$.
   - Discuss your findings.

3. 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:

   - $n \in \{1150\}$,
   - $p \in \{16\}$, and
   - `OMP_SCHEDULE` $\in$ { `static` | `static,1` | `dynamic,5` | `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 $n = 1150$, 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)

This 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. Parallelize the function `apply_filter` in `filter.c` using the `task` directive of OpenMP.
2. Measure the running time of your parallelized code for the following cases on one of the compute nodes of hydra:
   - $i \in \{\texttt{input1.png}\}$,
   - $p \in \{1, 2, 4, 8, 16, 24, 32\}$, and
   - $r \in \{1\}$ (The variable r denotes the number of rounds/times that the filter is applied.).

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

Option 2

Exercise 6 (22 [10 (code) + 4 (image scaling) + 4 (best grid/block size) + 4 (CPU vs GPU)] points)

In this exercise, you will compute the Julia set from the last assignment on a GPU using CUDA. We provide the CPU code for computing the Julia set using OpenMP. Your task is to convert the CPU code into a GPU version.

To that end, we have provided a source code template that can be used to either build the CPU or the GPU version of the Julia set application. You will only need to change `kernel_gpu.cu`, which defines the interface shown in Listing 4.

Listing 4: CPU/GPU Julia kernel interface.

```
void julia_kernel(float *julia_set, Complex c, float scale, int res_x, int
    res_y, int max_iter, float max_mag, float x_scale, float y_scale)
```

All parameters except `float *julia_set` are only needed for changing the way the Julia set is generated, but have no further impact on your task. Use the CPU version of the code as a reference. Note that all CUDA calls, e.g., `cudaMalloc()` should done from within this `julia_kernel` function or one of its sub-functions.

In particular, complete the following tasks:

1. Convert the CPU code to CUDA. One of the fundamental tasks is to determine a good default threads block size. Thus, if no thread block size is defined in the command line via `-blocksize`, determine a good default value of the block size.

2. Measure the running time of your CUDA program with one GPU on `tesla` for the following resolutions (for the tests, we only use always square images):
   - $r \in \{1000, 2000, 4000, 8000, 16000\}$.

   Here, use your default thread block size. Repeat the kernel launch fives times (`-nrep 5`).
   - Plot the average and maximum running time (y-axis) for each input size (x-axis).
   - Discuss your findings.

3. Now, fix the resolution to 20000×20000. Obtain the running time of the GPU code for the following thread block sizes and repeat each experiment five times (`-nrep 5`):
   - $b \in \{(1,1), (32,1), (1,32), (128,1), (1024,1)\}$.
   - Plot the average running time (y-axis) for each thread block size (x-axis).
   - Discuss your findings. Try to explain why we see or we do not see performance differences. To that end, use `nvprof` to see where the time has been spent.

4. Now, compare the running time of the CPU (using OpenMP) to the GPU. For different values of the image resolution, execute the CPU and the GPU code. Examine whether there is a cross-over point, i.e., a resolution from which one either the CPU code or the GPU code is outperforming the other. Use the following parameters:

- $r \in \{1000, 2000, 4000, 8000, 16000\}$,
- b, use the best block size from the previous experiment,
- $nrep \in \{5\}$.

Now provide the following information in your report:

- Plot the average running time (y-axis) for the CPU and the GPU code.
- Discuss your findings.

A. Additional Information for Option 1

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
```

On MacOS, you may use:

```
cmake -DCMAKE_C_COMPILER=/opt/local/bin/gcc-mp-13 .
```

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.

B. Additional Information for Option 2

B.1. Running CUDA code on `hydra/tesla`

In order to run CUDA code, you first have to connect via ssh to `hydra`. The compute node having GPU support is called `tesla`. Again, you have two options to run on `tesla`. You can either log in to `tesla` using `salloc` or you can submit a job using `sbatch`.

B.2. Compiling your CUDA code

To compile the CUDA code, you first need to login to `tesla`. Then, you can compile for this machine.

```
stester@hydra-head:~/bopc/julia_cuda$ pwd
/home/student/stester/bopc/julia_cuda

# starting an interactive job on tesla
stester@hydra-head:~/bopc/julia_cuda$ salloc -p q_student_gpu -t 10 --gres=gpu:tesla:1
salloc: Granted job allocation 259437
salloc: Waiting for resource configuration
salloc: Nodes tesla are ready for job

stester@tesla:~/bopc/julia_cuda$ nvidia-smi -L
GPU 0: Tesla T4 (UUID: ..)
GPU 1: Tesla T4 (UUID: ..)

# compiling the GPU code
stester@tesla:~/bopc/julia_cuda/src$ make -f Makefile.gpu
nvcc -DMODE=1  -x cu -c juliaset.cpp
nvcc -DMODE=1  -c kernel_gpu.cu
nvcc -DMODE=1  -o juliaset_gpu juliaset.o image.o cli_parser.o kernel_gpu.o

# compiling the CPU code
stester@tesla:~/bopc/julia_cuda/src$ make clean
rm *.o

stester@tesla:~/bopc/julia_cuda/src$ make
g++ -fopenmp -c juliaset.cpp
g++ -fopenmp -c image.cpp
g++ -fopenmp -c cli_parser.cpp
g++ -fopenmp -c kernel_cpu.cpp
g++ -fopenmp -o juliaset_cpu juliaset.o image.o cli_parser.o kernel_cpu.o
```

B.3. Running Binaries Interactively

If you are logged into `tesla`, you can simply execute your code in your current shell.

```
# this is the empty function (that is why the runtime is 0)
stester@tesla:~/bopc/julia_cuda/src$ ./juliaset_gpu -r 100 100 -n 4
0;100;100;1.5;-1;-1;0
1;100;100;1.5;-1;-1;0
2;100;100;1.5;-1;-1;0
3;100;100;1.5;-1;-1;0

stester@tesla:~/bopc/julia_cuda/src$ ./juliaset_cpu -r 100 100 -n 4
0;100;100;1.5;-1;-1;16515
1;100;100;1.5;-1;-1;18175
2;100;100;1.5;-1;-1;6778
3;100;100;1.5;-1;-1;12719
```

B.4. Running Binaries Using `sbatch`

You can also enqueue batch files from `hydra`. Note that you will need to compile your code first on `tesla`. Then, you can simply add your batch file to the SLURM queue, like this

```
stester@hydra-head:~/bopc/julia_cuda/jobs$ sbatch gpu_code.job
Submitted batch job 259438
```