ETH zürich

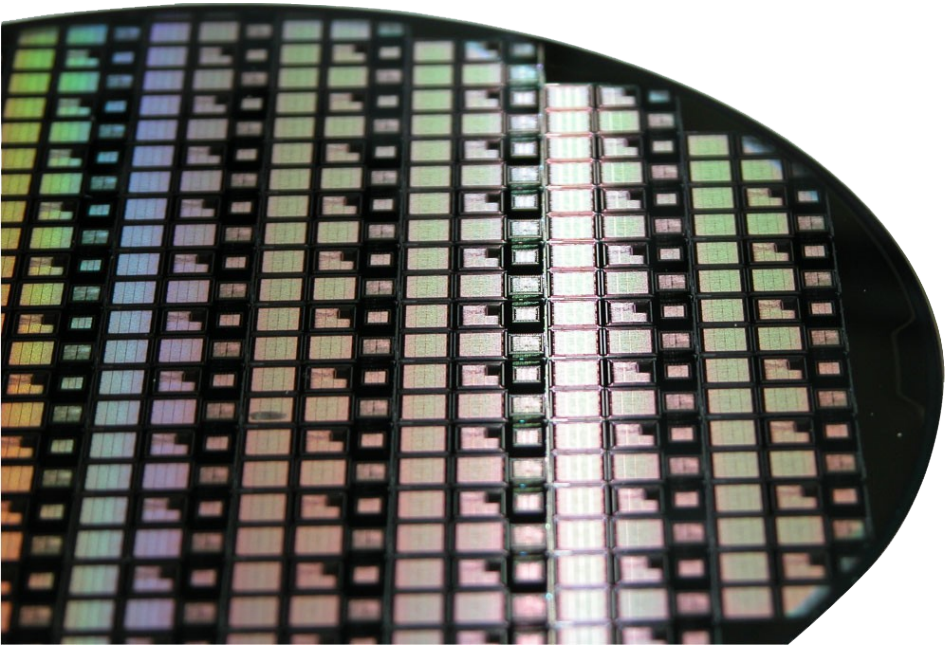
Exercise: OpenMP Programming

Multicore programming with OpenMP

26.03.2024

Philip Wiese – *wiesep@iis.ee.ethz.ch*

Zexin Fu – *zexifu@iis.ee.ethz.ch*



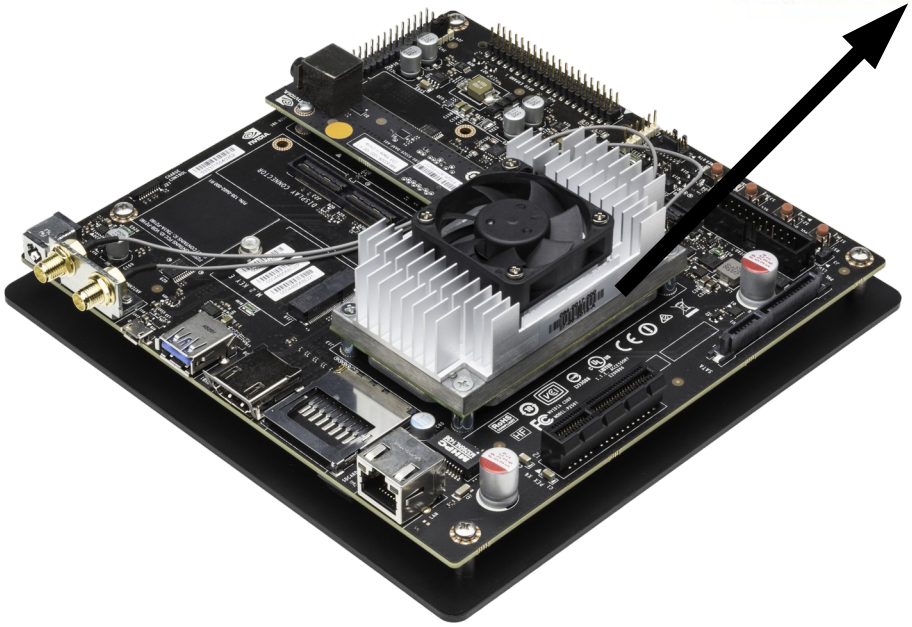
ETH zürich

Integrated Systems Laboratory

Exercise description

- This exercise consists of four parts:
  - Ex 1, 2, 3: Introduction to OpenMP and parallel programming fundamentals
  - Ex 4, 5, 6: Data parallelism, OpenMP loop scheduling
  - Ex 7: Parallelisation of Convolution kernel
  - Ex 8-9: Task parallelism, OpenMP sections and tasks
- Freedom at your own responsibility:
  - If you are unsure about the fundamentals, make sure that you understand exercise 1-3, as you might otherwise have problems to finish future exercises.
  - If you are confident in parallel programming fundamentals, you can skip exercise 1-3 and spend time on the advanced topics (Ex 4-9). Note however that all topics may appear as questions in the final examination.

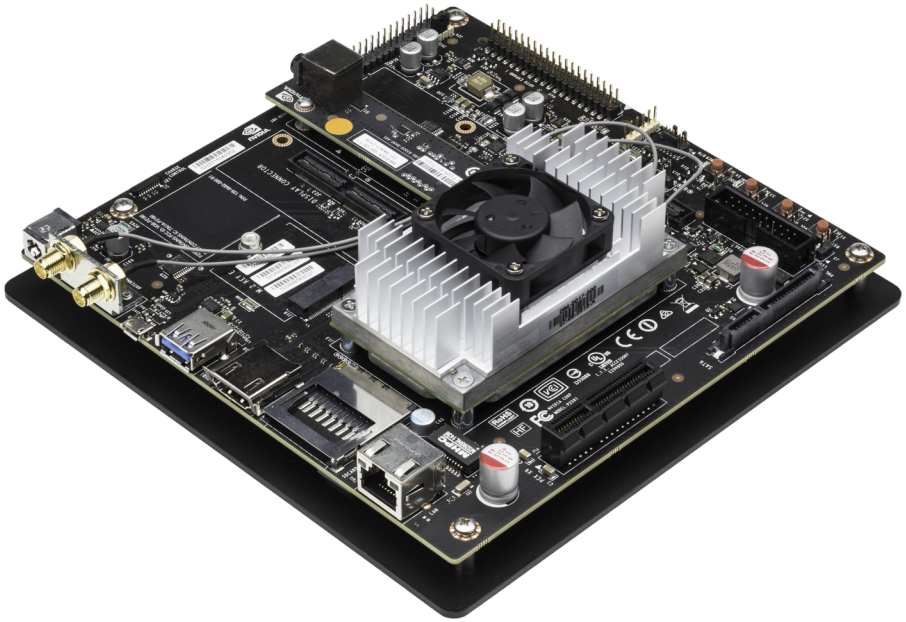
Hardware used: NVIDIA Tegra TX1



	JETSON TX1
GPU	1 TFLOP/s 256-core Maxwell
CPU	64-bit ARM A57 CPUs
Memory	4 GB LPDDR4 | 25.6 GB/s
Storage	16 GB eMMC
Wifi/BT	802.11 2x2 ac/BT Ready
Networking	1 Gigabit Ethernet
Size	50mm x 87mm
Interface	400 pin board-to-board connector

In this exercise we will only program the 4-core CPU. In the next exercise, you will also program the GPU.

Board Assignment



You will work in pairs of two students.

Please fill in the board assignment sheet

https://shorturl.at/qrsxR

Login, compile and run

- Login to TX1 board:

```
ssh student@tx1-##.ee.ethz.ch
```

- Run the maxPerf script to disable DVFS scaling:

```
sudo ./maxPerf.sh
```

- Get the exercise:

```
~/socdaml/setup-multicore-fs24-ex.sh
```

- The exercise code skeleton is now available in:

```
~/socdaml/ex_multicore/
```

You can re-run the setup script if you need a fresh copy.

Login, compile and run

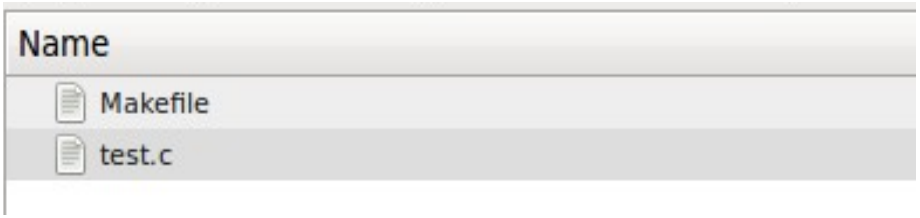
- Compile and run with

  `make clean all run`

- Take a look at **`test.c`**. Different exercises are `#ifdef`-ed

- **To compile and execute the desired exercise:**

  `make clean all run –e MYOPTS="-DEX1 –DEX2 …"`



Ex 1 – Parallelism creation: Hello World!

```
#pragma omp parallel num_threads (?)
  printf "Hello world, I'm thread ??"
```

- Use parallel directive to create multiple threads
  - Each thread executes the code enclosed within the scope of the directive
- Use runtime library functions to determine thread ID
  - All SPMD parallelization is based on this approach

Ex 2 – Racing and Synchronization

- A correct sequential algorithm is not necessarily correct if executed in parallel.
- For example, if one thread sets *x* to *1* and a second thread sets *x* to *5* at the same time, what *x* will a third thread see?
- This is called a **data race** and we use **synchronization** to avoid it.
- One important synchronization primitive is the **critical section:**

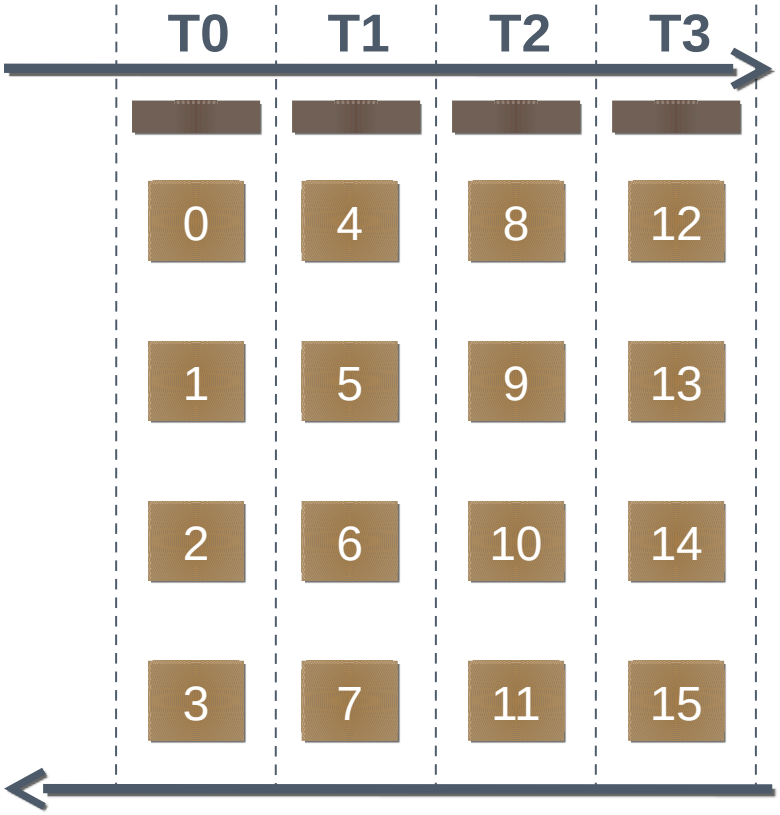
```
#pragma omp critical
{
  // code executed by one thread at a time
}
```

- Complete the code in the exercise.

Ex 3 – Deadlocks

- A correct sequential algorithm does not necessarily terminate if executed in parallel.
- We call the situation in which all threads wait for another thread to continue a **deadlock**.
- Deadlocks can occur when parallelization introduces dependencies between threads – and it is the **responsibility of the programmer** to **prevent** them!
- Complete the code in the exercise
- *Hint:* You can terminate a running program in the shell with *Ctrl+C*
- *Hint:* This is called the “Dining philosopher problem”

Ex 4 – Loop partitioning: Static scheduling



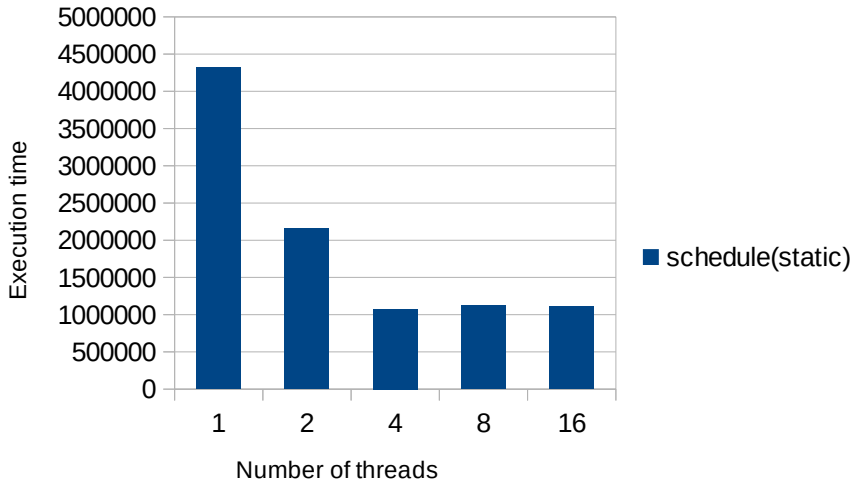

```
#pragma omp parallel for \
    num_threads (NTHREADS) schedule (static)

for (uint i=0; i<NITERS; i++)
{
  work (W);

} /* (implicit) SYNCH POINT */
```

- With **`schedule(static)`** M iterations are statically assigned to each thread, where M = NITERS / NTHREADS
- **Small overhead**: loop indexes are computed according to thread ID
  - Optimal scheduling if workload is balanced

Ex 4 – Loop partitioning: Static scheduling

Exercise 4a

```
#pragma omp parallel for \
    num_threads (NTHREADS) schedule (static)

for (uint i=0; i<NITERS; i++)
{
  work (W);

} /* (implicit) SYNCH POINT */
```

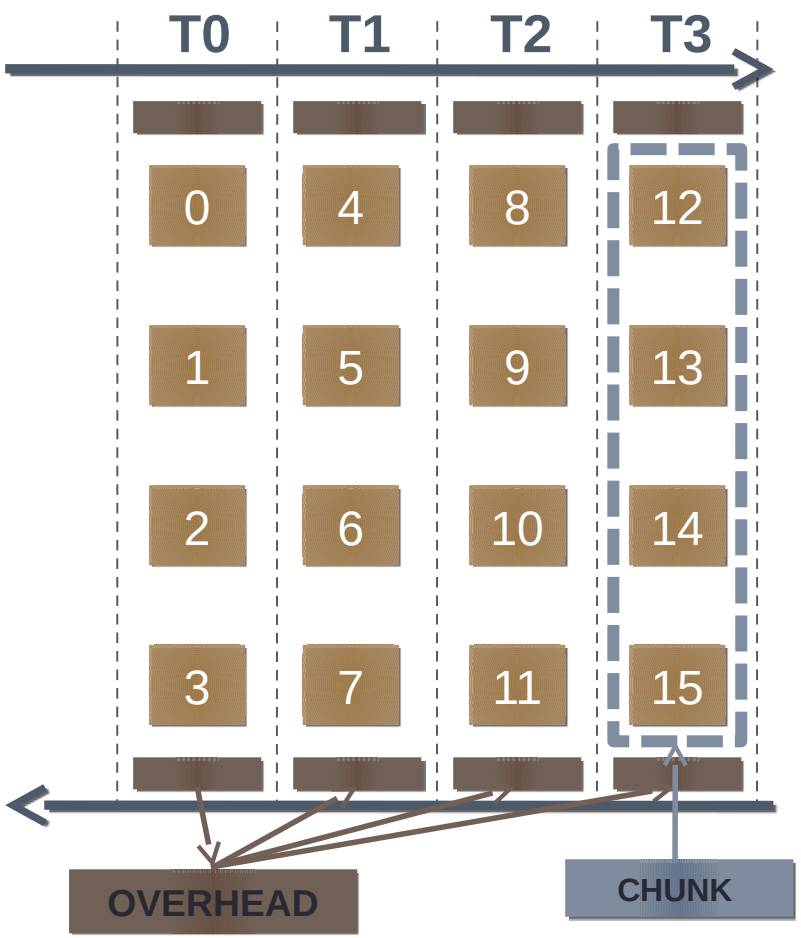


Use the following parameters:

- NITERS = 1024
- NTHREADS = {1,2,4,8,16}
- W = 1000000

- Collect execution time in excel sheet for the various configurations
- Comment on the results

Ex 4 – Loop partitioning: Dynamic scheduling

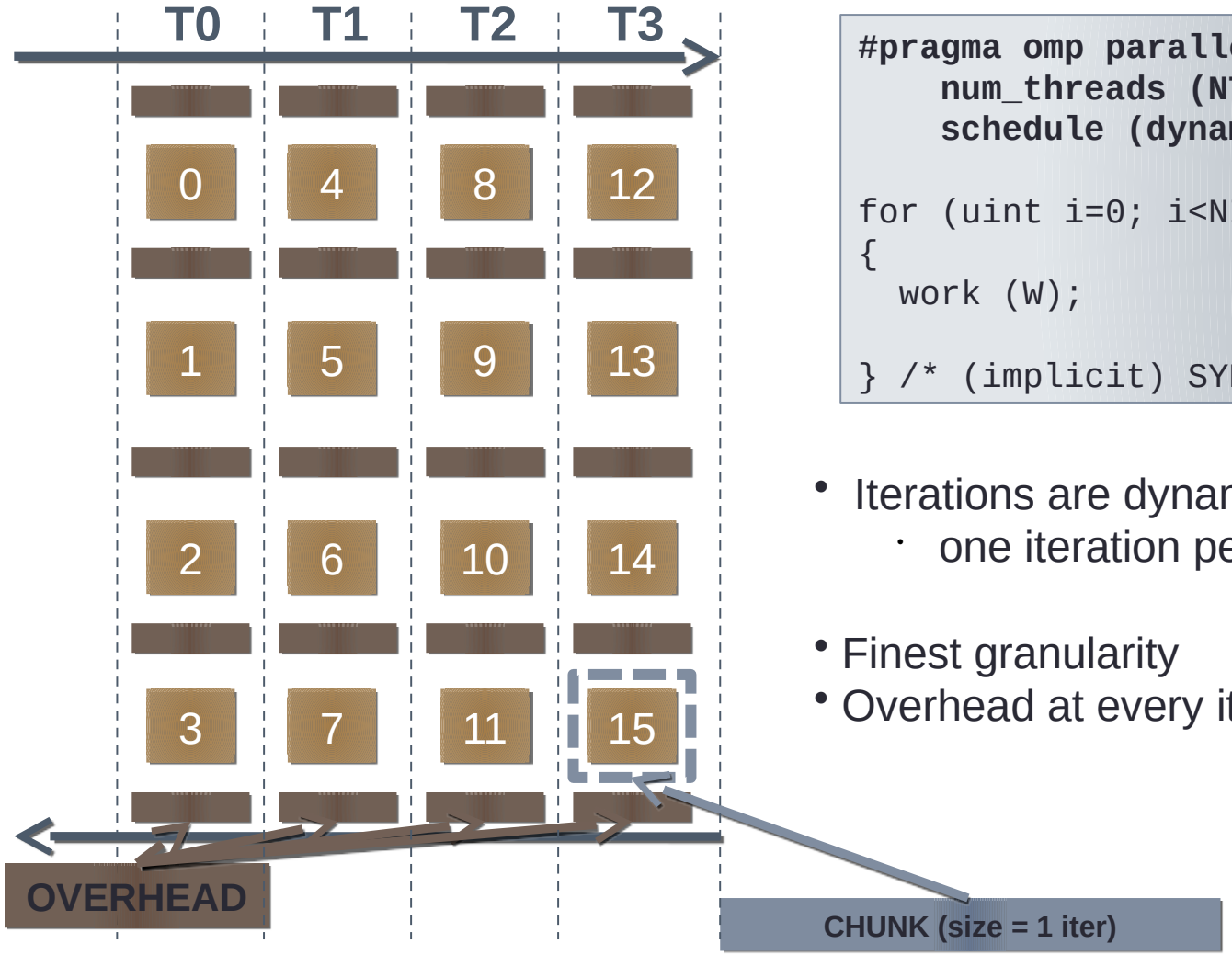



```
#pragma omp parallel for        \
    num_threads (NTHREADS)      \
    schedule (dynamic, M)

for (uint i=0; i<NITERS; i++)
{
  work (W);

} /* (implicit) SYNCH POINT */
```

- Iterations are dynamically assigned to threads in groups of M = NITERS/NTHREADS

- Same parallelization of **`schedule(static)`**
- Coarse granularity
- Overhead only at beginning and end of loop

Ex 4 – Loop partitioning: Dynamic scheduling



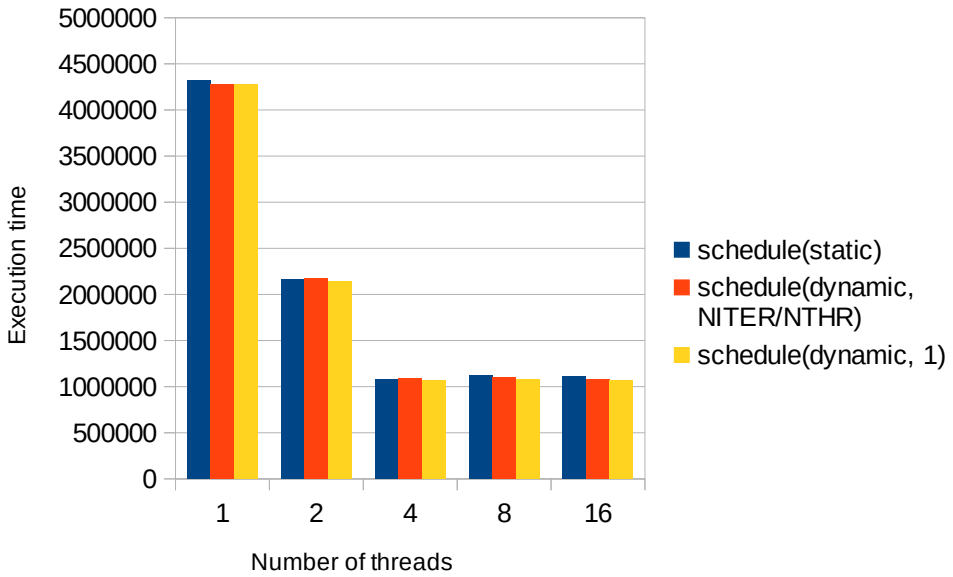

```
#pragma omp parallel for       \
    num_threads (NTHREADS)     \
    schedule (dynamic, M)

for (uint i=0; i<NITERS; i++)
{
  work (W);

} /* (implicit) SYNCH POINT */
```

- Iterations are dynamically assigned to threads
  - one iteration per chunk
- Finest granularity
- Overhead at every iteration

Ex 4 – Loop partitioning: Dynamic scheduling

Exercise 4b

```
#pragma omp parallel for        \
    num_threads (NTHREADS)      \
    schedule (dynamic, M)

for (uint i=0; i<NITERS; i++)
{
  work (W);

} /* (implicit) SYNCH POINT */
```



Use the following parameters:

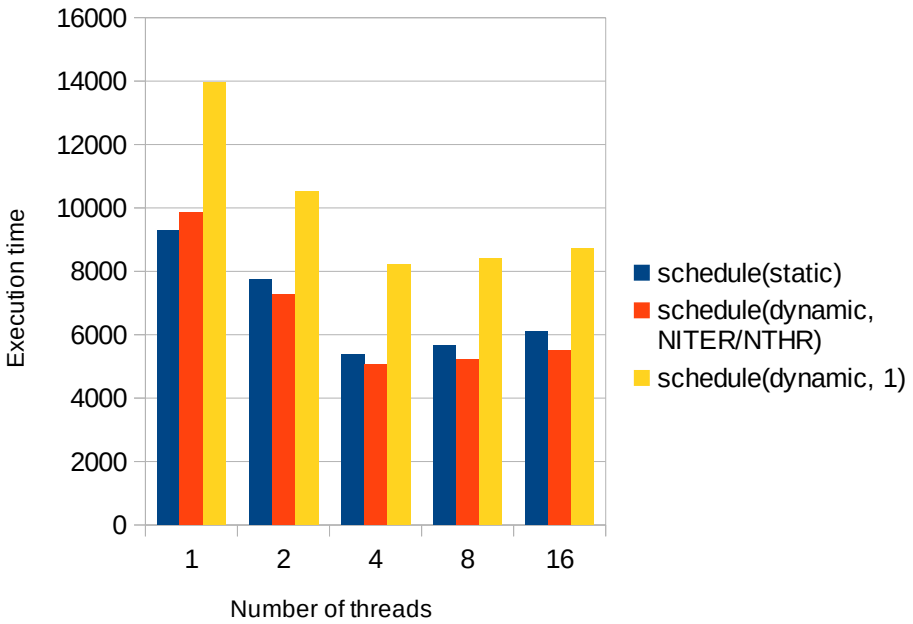
- NITERS = 1024
- NTHREADS = {1,2,4,8,16}
- M = {NITERS/NTHREADS, 1}
- W = 1000000

- Add execution time for the various configurations to the excel sheet
- Comment on the results

Ex 4 – Loop partitioning: Dynamic scheduling

Exercise 4c

```
#pragma omp parallel for      \
    num_threads (NTHREADS)    \
    schedule (dynamic, M)

for (uint i=0; i<NITERS; i++)
{
  work (W);

} /* (implicit) SYNCH POINT */
```

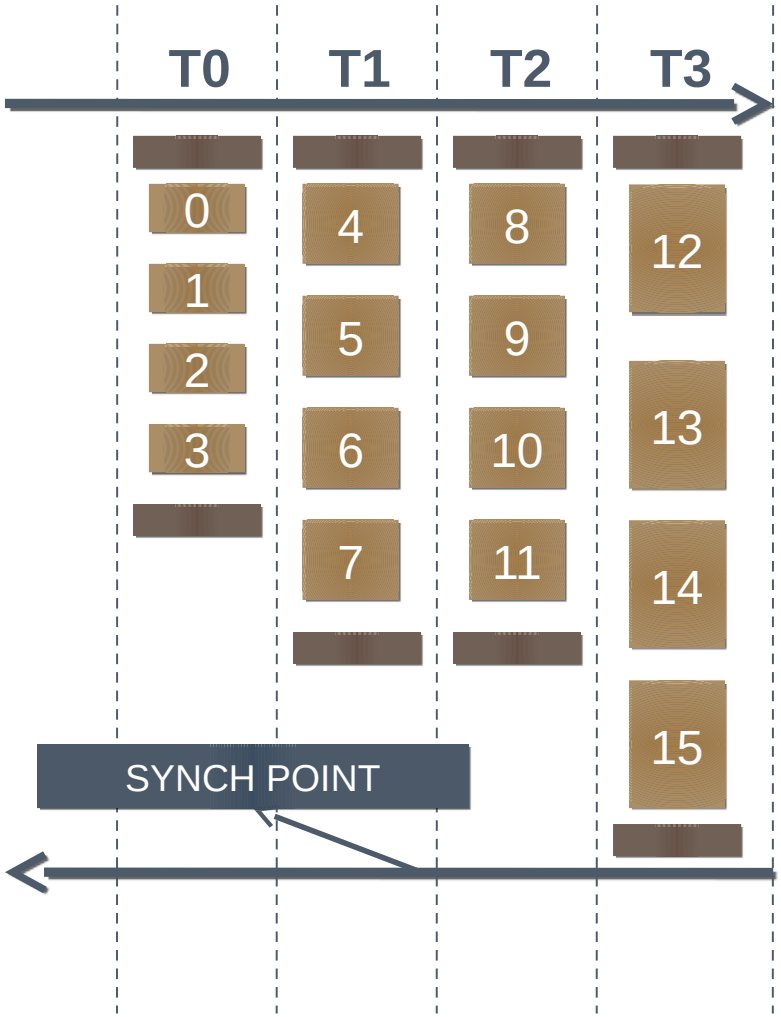


Use the following parameters:

- NITERS = **102400**
- NTHREADS = {1,2,4,8,16}
- M = {NITERS/NTHREADS, 1}
- W = **10**

- Draw the same plot as exercise 2a, 2b
- What happens for smaller workload and higher iteration count?

Ex 5 – Unbalanced Loop Partitioning

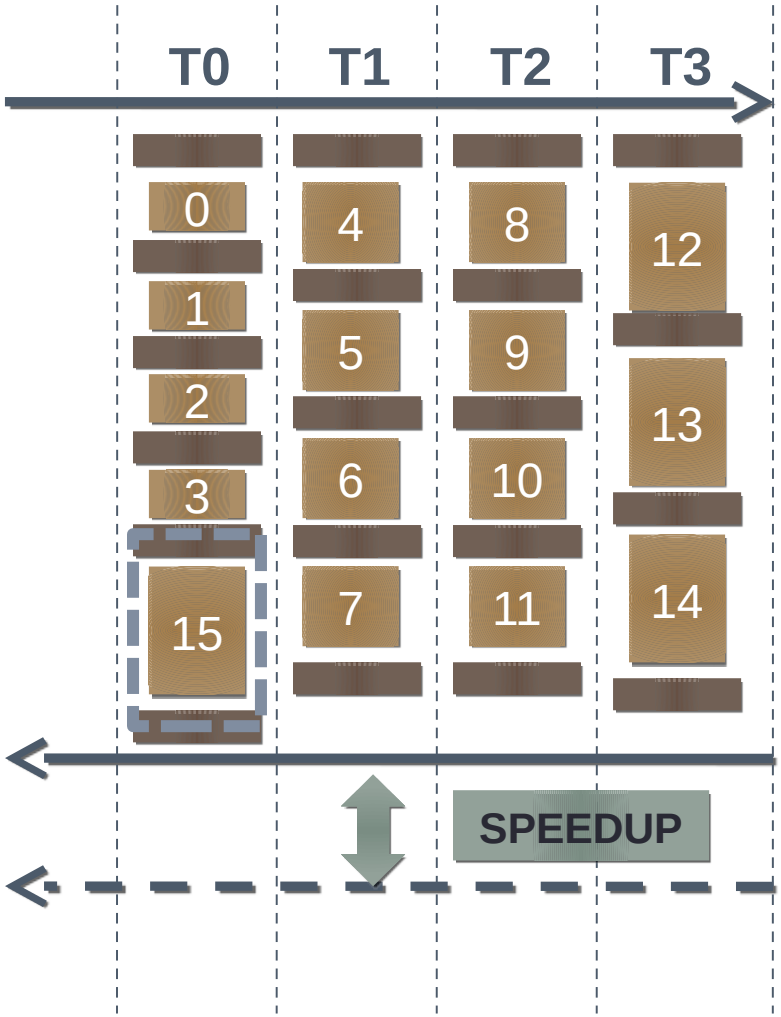



```
#pragma omp parallel for      \
    num_threads (NTHREADS)    \
    schedule (dynamic, NITERS/NTHREADS)

for (uint i=0; i<NITERS; i++)
{
  work((i>>2) * W);

} /* (implicit) SYNCH POINT */
```

- Iterations have different duration
- Using coarse-grained chunks (NITERS/NTHREADS) creates unbalanced work among the threads

- Due to the barrier at the end of parallel region, all threads have to wait for the slowest one

Ex 5 – Unbalanced Loop Partitioning




```
#pragma omp parallel for        \
    num_threads (NTHREADS)      \
    schedule (dynamic, 1)

for (uint i=0; i<NITERS; i++)
{
  work((i>>2) * W);

} /* (implicit) SYNCH POINT */
```

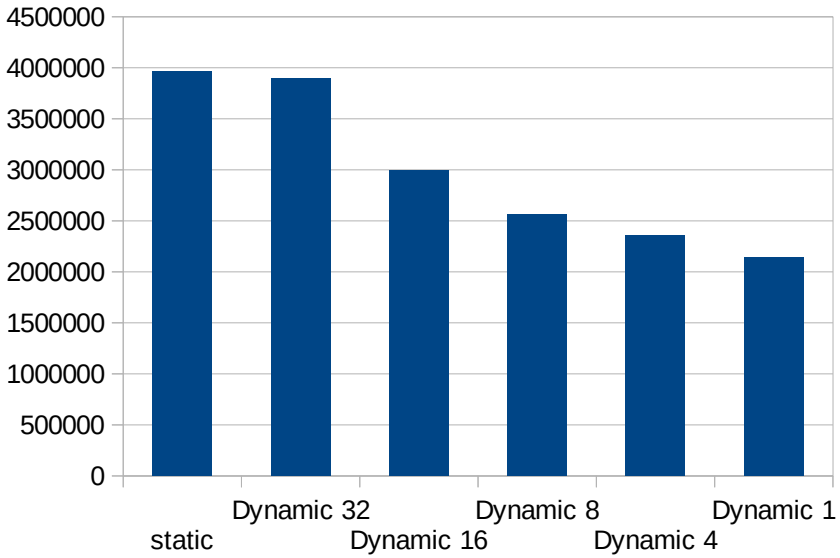
- Iterations have different duration
- Using fine-grained chunks (the finest is 1) creates balanced work among the threads

- The overhead for dynamic scheduling is amortized by the effect of **work-balancing** on the overall loop duration

Ex 5 – Unbalanced Loop Partitioning

Exercise 5a, 5b

```
#pragma omp parallel for      \
    num_threads (NTHREADS)    \
    schedule (dynamic, M)

for (uint i=0; i<NITERS; i++)
{
  /* BALANCED LOOP CODE */

} /* (implicit) SYNCH POINT */
```



Use the following parameters:

- NITERS = **128**
- NTHREADS = 4
- M = {32, 16, 8, 4, 1}
- W = **1000000**

- Create a new excel sheet plotting results (execution time) for static and dynamic schedules (with chunk size M)
- Comment on results

Ex 6 – Chunking Overhead

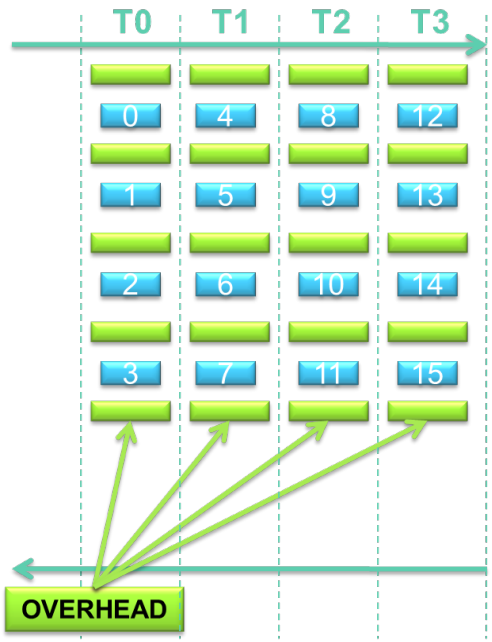
- Study the impact of chunk size for varying sizes of the workload W

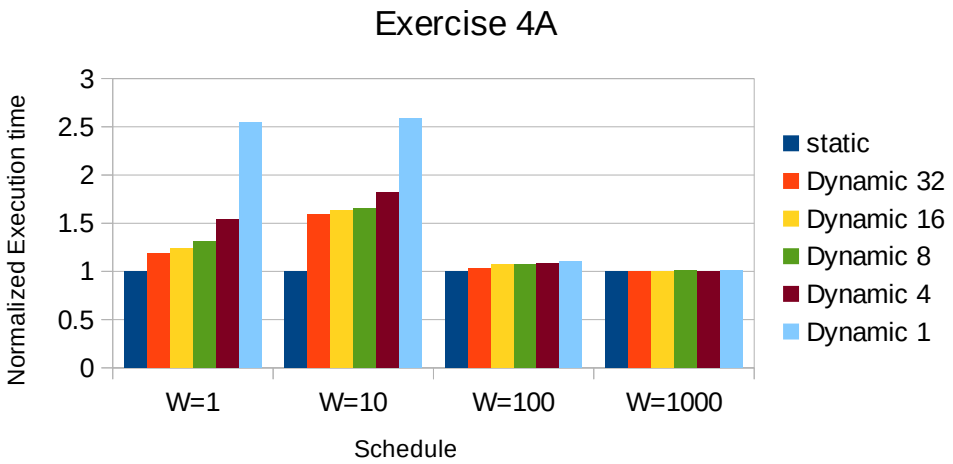
Use the following parameters:

- NITERS = **1024*256**
- NTHREADS = 4
- M = {32, 16, 8, 4, 1}
- W = **{1, 10, 100, 1000}**

- Create a new excel sheet plotting results (execution time) for static and dynamic schedules (with chunk size M)
- Comment on results

***HINT**: Plot NORMALIZED execution time to the fastest scheduling option for a given value of W*





Ex 7 – Parallelizing 5x5 Convolution

Let's have a look at the **Conv5x5_Scalar** kernel that you worked with in the previous exercise.

- How can Conv5x5_Scalar be parallelized with OpenMP? Add the relevant pragma to the code.
  - This is loop based code, so we will probably want to go with a `parallel for`. There are many loops to choose from, which one should we select? Is this workload balanced or unbalanced? Select a reasonable schedule.
  - Assign the correct **data sharing** (`shared`, `private`, `firstprivate`, ...) to ensure that each thread is working on the right data. Hint: By adding default(none) to the pragma, OpenMP will force you to explicitly define data sharing for each variable. This is useful to make sure you don't miss assigning a data sharing attribute.
  - **Race conditions:** Does your selected parallelization method cause any race conditions?

- Run the OpenMP kernel a few times. Do you always get the correct result? If you do, explain what part of the OpenMP pragma ensures this. If you don't get the right results, you should refine your solution.

- Compare the execution time of the code with 1, 2, and 4 threads. Use Amdahl's Law to calculate how large portion of the code that is parallelized. Can we increase this part?

Ex 8 – Task parallelism with `sections`

```
void sections()
{
      work(1000000);
      printf("%hu: Done with first elaboration!\n”, …);
      work(2000000);
      printf("%hu: Done with second elaboration!\n”, …);
      work(3000000);
      printf("%hu: Done with third elaboration!\n", …);
      work(4000000);
      printf("%hu: Done with fourth elaboration!\n", …);
}
```

a) Distribute workload among 4 threads using SPMD parallelization
   I. *Get thread id*
   II. *Use if/else or switch/case to differentiate workload*

b) Implement the same workload partitioning with `sections` directive

Ex 9 – Task parallelism with `task`

```
void tasks()
{
  unsigned int i;

  for(i=0; i<4; i++)
  {
        work((i+1)*100000);
        printf("%hu: Done with elaboration\n", …);
  }
}
```

a) Distribute workload among 4 threads using **`task`** directive
   a) *Same program as before*
   b) *But we had to manually unroll the loop to use sections*
   c) *Add some prints to obtain the following output :*

b) Can you reproduce this using **`sections`**?

```
      (...)
0: Spawning of 309
0: Spawning of 310
0: Spawning of 311
0: Spawning of 312
2: Done with 53
1: Done with 55
3: Done with 54
2: Done with 56
1: Done with 57
      (...)
```