TDT4200 Parallel programming

PS4

Maren Wessel-Berg & Claudi Lleyda Moltó

October 2023

Practical information

Published: 03/10/23

Deadline: 10/10/23 at 22:00

Evaluation: pass/fail

- Completing the problem set is **mandatory**.
- The work must be done **individually** and without help from anyone but the TDT4200 staff.
- **Reference** all sources found on the internet or elsewhere.
- The **requirements**, and **how and what to deliver** is explained in the problem set description found on BlackBoard.
- Start the exercises early!

Where can you get help with the assignment?

- **Recitation lecture**: introduction to the problem set
  (Today)
  Slides will be made available online.
- **TA hours**: ask questions in person
  Friday, October 6th, 10:00–12:00 in Cybele
  Monday, October 9th, 13:00–15:00 in Cybele
- **Piazza**: question forum
  Ask questions any time (but give us time to answer).
  Select the ps4 folder for questions related to this problem set.
  Do not post full or partial solutions!

pthreads and OpenMP

- We will discuss two approaches for parallelizing code for a single machine.
  - pthreads is a low-level API that allows us to spawn and manage threads from within a process.
  - OpenMP is a high-level interface for parallelizing code, providing many useful portable constructs.
- We can easily share the computation domain among threads, meaning we do not need to worry about transferring the border across processes!

Programming with pthreads

Thread handler

- The type `pthread_t` is an abstract thread handler, storing a unique thread identifier.
- We need one instance for each simultaneous thread that we want to launch.
- pthreads will use the instances of `pthread_t` to orchestrate the threads.

Creating and joining threads

Starting a thread

- We can start a new thread by calling `pthread_create`.
  - `pthread_t *thread`, a pointer to a thread handler. This will create a new, currently unused, thread ID.
  - `pthread_attr_t *attr`, a pointer to a structure determining thread attributes. We can ignore it by setting it to `NULL`.
  - `void *(*start_routine)(void *)`, a pointer to a function that takes in a `void *` and returns a `void *`. The created thread will immediately run this function with the input provided in the next argument, `arg`.
  - `void *arg` Input to the provided start routine in the previous argument, `start_routine`.

Joining threads

- We can wait for a thread to terminate by calling `pthread_join`.

Working with threads

```c
#include <pthread.h>
#include <stdio.h>
#define N 17

void *func(void *arg) {
    char *c = arg;
    printf("%c", *c);
    return NULL;
}

int main(void) {
    pthread_t th[N];
    char str[N + 1] = "Race␣condition␣:(";
    for(int i = 0; i < N; i++)
        pthread_create(th + i, NULL, func, str + i);
    for(int i = 0; i < N; i++)
        pthread_join(th[i], NULL);
    printf("\n");
}
```

Synchronizing threads

Thread barriers

- We can initialize a `pthread_barrier_t` by calling `pthread_barrier_init`, and specifying a count.
- A `pthread_barrier_t` will stop threads that wait on it by calling `pthread_barrier_wait`, and keep track of how many are waiting, until the predefined count has been reached.
- We can destroy a barrier with `pthread_barrier_destroy` when we are done using it.

Programming with OpenMP

- OpenMP is a high-level interface for writing parallel code.
- We use `#pragma omp` directives to tell the compiler where to parallelize our code.
  - We can use `#pragma omp parallel` to instruct OpenMP to use as many threads as it wants.
  - We can use `#pragma omp for` to parallelize a loop.

Example

```
#pragma omp parallel for
for(int i = 0; i < N; i++)
    a[i] = b[i] + c[i];
```

Parallel regions

- We can avoid creating and destroying threads for every `#pragma omp` instruction unnecessarily.
- Just encase multiple lines of code in a block.

Example

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
        a[i] = b[i] * c[i];

    #pragma omp for
    for(int i = 0; i < N; i++)
        c[i] = a[i] + b[i];
}
```

Reduction

- When performing a reduction operation, we can tell OpenMP how to parallelize it.
- We must specify what operation we are using, and on what variable we are reducing.

Example

```
#pragma omp parallel for reduction (+ : r)
for(int i = 0; i < N; i++)
    r += a[i];
```

Barriers

- We can also use barriers in OpenMP code!
- We just need to signal where we want the barrier to occur.

Example

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
        a[i] += b[i];

    #pragma omp barrier

    #pragma omp for
    for(int i = 0; i < N; i++)
        b[i] += a[N-i];
}
```

Master thread

- We can specify an instruction to only be executed by the master thread.

Example

```
#pragma omp parallel
{
    #pragma omp for
    for(int i = 0; i < N; i++)
        a[i] += b[i];
    #pragma omp master
    write_to_file(a);

    #pragma omp for reduction (+ : sum)
    for(int i = 0; i < N; i++)
        sum += a[i];
    #pragma omp master
    printf("%d\n", sum);
}
```