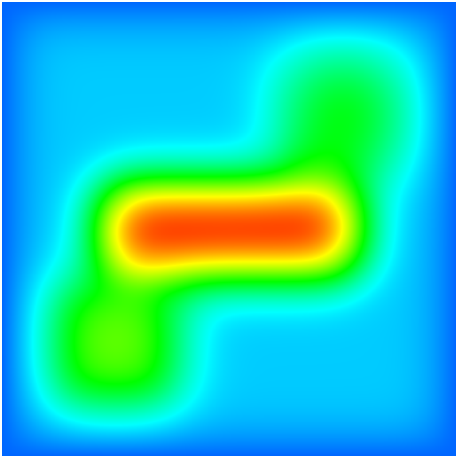
Recitation 3

Announcements

- Assignment 3 is out, dealine 05.10 (for both code and theory):

Heat equation



We’ll parallelize the same computation as in the previous assignment, this time using OpenMP and pthreads.

Parallelization

- In more detail, you should parallelize the double for loops in `ftcs_solver()` and `external_heat()`
- NB you did not touch `external_heat()` in the previous assignment, but need to parallelize it in this.
- Each iteration of the loops is independent of the others. We can therefore do the parallelization by dividing the iterations evenly between the threads.
- Since the iterations of the loops correspond to rows/columns, we're dividing the grid among the threads, similar to what we did in the previous assignment.
- However, since we're using threads and shared memory, there is only one `temperature` array, with the different threads working on differnt parts. We don't need to scatter/gather/do border exchanges.

Parallelization

- A simple parallelization would start several threads at the beginning of double loops in `ftcs_solver()` and `external_heat()`, and kill them at the end of the for loop.
- `ftcs_solver()` and `external_heat()` are called once for each iteration of the time loop in the `main()` function.
- This scheme would therefore cause threads to be spawned/killed twice for each iteration of the time loop in main, with causes some overhead.
- Better, but more complex, solution is therefore to launch all the threads before the time loop, use them as needed in the time loop, and kill them at the end.
- If you do this, keep in mind that the output should only be done by a single thread.

Details

- Just like we had two `local_temp` arrays in the previos assignment, we have two `temperature` arrays.
- To make it easier to deal with the global boundary condition, these have a border/halo of 1.
- The `ti()` function can be used to index the `temperature` array like in the previous assignment, taking the border into account.
- All of this is allready done in the serial version. You will typically not need to change the body of the for loops in `ftcs_solver()` or `external_heat()`.

Your task

- You’ll be given a serial implementation.
- You should create two parallel implementations:
    - One using pthreads
    - One using OpenMP
- Your program should work for any number of threads (including numbers not a power of two), read from the command line.

Testing

- The serial version produces the correct output, the parallel versions should produce the same output.
- Remember that if you have a race condition/potential deadlock, you can get the correct output if you’re lucky even if your program is incorrect.

Minimal pthreads

```
#include <stdio.h>
#include <pthread.h>

void* run_thread(void* arg){
  printf("Thread␣%ld\n", (long)arg);
  pthread_exit(NULL);
}

int main(){
  pthread_t threads[4];
  for(long i = 0; i < 4; i++){
    pthread_create(&threads[i], NULL,
      run_thread, (void*)i);
  }
  pthread_exit(NULL);
}
```

Passing arguments to threads

```
typedef struct{
  int arg1;
  int arg2;
} args;
...
void* run_thread(void* a){
  args* b = (args*)a;
  printf("%d\n", b->arg1)
  ...

main(){
  ...
  args* a = malloc(...
  a->arg1 = 4;
  a->arg2 = 5;
  pthread_create(..., (void*)a);
```

Minimal OpenMP

```
#include <stdio.h>
#include <omp.h>

int main(){

#pragma omp parallel for
  for(int c = 0; c < 8; c++){
    printf("Thread:␣%d␣of␣%d\n",
      omp_get_thread_num(),
      omp_get_num_threads());
  }
}
```