Chair for Computational Analysis of Technical Systems

RWTH AACHEN UNIVERSITY

Parallel Computing for Computational Mechanics

Summer semester 2021

Homework 3: Shared Memory Parallelism with OpenMP

---

1 Introduction

In this homework, we modify the 2D unsteady heat equation solver from Homework 2 by implementing compiler optimization flags and by parallelizing time consuming loops using the OpenMP framework.

2 Task Description

This homework consists of 5 tasks:

1: **Task 1:** Investigate the effect of different compiler flags in `CMakeLists.txt` which is located in the `skeleton` folder. Record the execution time of your code with a reduced number of timesteps on the coarse mesh and compare the timing for different compiler flag options. Remember the flags from the class participation session.

2: **Task 2:** Identify the most time consuming loops within the FE solver. Typically, parallelizing these loops will yield in a speedup of your code. Implement a timer around each loop that you want to inspect and compare the execution times of those loops. For the latter, we recommend the usage of `omp_get_wtime()` from the very beginning. Please adjust the timings inside the file `2D_Unsteady_Diffusion.cpp` accordingly.

3: **Task 3:** Choose the two most time-consuming loops and parallelize them using OpenMP statements. When relevant, consider data races appropriately. For this, use the two most common approaches to avoid a race condition, i.e., `omp critical` and `omp reduction`. Which of these options performs best and why? To avoid unnecessary use of CPU time and efficient debugging, we recommend you to use the coarse mesh initially.

4: **Task 4:** For the loop parallelization, try the scheduling options discussed during the lecture (`static`, `dynamic`, and `guided`). Report on the differences you observe for the different options and discuss why these differences occur?

5: **Task 5:** Execute the code via the batch system using 1, 2, 4, 8, and 12 cores. To submit a batch job, use the `run.j` script in Listing 3. Record the timings for the different meshes and number of cores. Calculate the parallel speed-up and efficiency for your OpenMP implementation. With respect to the different mesh refinements and number of cores does OpenMP parallelization pay off?

3 Hints for the Tasks

3.1 General Hints

We recommend carrying out this homework on the ITC cluster. To use OpenMP, you have to embed the OpenMP library in the header of the file with:

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

To efficiently debug your code, we recommend using the coarsest mesh and optionally you can reduce the number of time steps in the settings file. Remember to set the number of iterations back to the default when evaluating the parallel performance.

3.2 Task 1: Compiler optimization

The main routine in `2D_Unsteady_Diffusion.cpp` already contains a timing functionality. You can read the timing from the output file of the solver. Make your modifications to `CMakeLists.txt` in line 4 (`add_compile_options(-g)`). When compiling, make sure to choose a suitable compiler for this code. To import the path to the correct compilers, type the following lines from Listing 1 into your terminal:

Listing 1: Intel Compiler Options

```
module load DEVELOP
module load intel/19.0
export CXX=`which icpc`
export C=`which icc`
```

3.3 Task 2: Identifying the most time consuming loops

You can use the OMP timing function `omp_get_wtime()`. The pseudo code in Listing 2 illustrates its usage. When choosing the loops, think about what can cause a certain loop to be time consuming. Also think about which loops can be parallelized and which loops are not suitable for parallelization.

Listing 2: Timing with OpenMP

```
double startIter, endIter, timeIter;
startIter = omp_get_wtime(); \\Start the timer.
for (int i=0; i<iMax; i++) \\Some loop that you want to time.
{}
endIter = omp_get_wtime(); \\Stop the timer.
timeIter = endIter-startIter ; \\Calculate the time in the loop.
```

3.4 Tasks 3 and 4: Loop parallelization

List all variables in the loop and think about whether they have to be shared or private. If multiple threads have to access a memory entry, this is an indicator that there might be a data race. Implement the critical and reduction clauses to prevent the data race. Which one do you expect to perform better and why? For

the scheduling, think about whether the computational load per iteration changes or not. What is the default scheduling type in OpenMP?

3.5 Task 5: Running on multiple cores

To run the solver in parallel using OpenMP, you can use the `run.j` script in Listing 3. In this script, change the number of cores by setting `#SBATCH -cpus-per-task`.

```
#!/usr/bin/env bash
### Lecture queue.
#SBATCH --account=lect0051

### Ask for exactly one node -> all allocated CPUs must be on this one.
#SBATCH --nodes=1

### Ask for <1 or 2 or 4 or 6 or 8 or 12> cpus.
#SBATCH --cpus-per-task=<1 or 2 or 4 or 6 or 8 or 12>

### Divide the needed memory per task through the cpus-per-task,
### as slurm requests memory per cpu, not per task!
### Example:
### You need 2 GB memory per task, you have 8 cpus per task ordered
### order 2048/ <1 or 2 or 4 or 6 or 8 or 12> ->
### <2048M or 1024M or 512M or 342M or 256M or 171M> memory per task.
### M is the default and can therefore be omitted,
### but could also be K(ilo)|G(iga)|T(era).
#SBATCH --mem-per-cpu=<2048M or 1024M or 512M or 342M or 256M or 171M>

### Name of the job.
#SBATCH --job-name=2d_Unsteady_OpenMP

### Outputs of the job.
#SBATCH --output=out.%j.txt
#SBATCH --error=err.%j.txt

### Wall clock limit.
#SBATCH --time=0:30:00

### Run the process
../build/2d_Unsteady_OpenMP ./settings.<coarse or fine or finest>.in
```

Listing 3: Run script for the SLURM batch system on the RWTH cluster

If you choose more CPUs and the problem size stays the same, the memory required per CPU will decrease. Hence, you also need to adapt the required memory accordingly by using `#SBATCH -mem-per-cpu`. In the last line of the batch script the solver is executed. Here you can specify what input file the solver should use.Plot your timings, speed-up, and parallel efficiency data as a function of the number of CPUs for the different meshes. To compute relative quantities, you can use the data of the serial solver. Also, plot the numerical solution obtained with the serial and parallelized code to verify your implementation.

IMPORTANT: The exclusive flag, `"#SBATCH --exclusive"`, is meant to be used to collect timings, but not for testing or debugging. Make sure it is set correctly when collecting runtime data. Please use computing resources conscientiously. Computations on large numbers of CPUs should performed when you are certain your code is working correctly.

Contact:
Maximilian Schuster · schuster@cats.rwth-aachen.de
Michel Make· make@cats.rwth-aachen.de