# Exercises Lecture Computational Physics (Summer 2011)

Lecturers: Volker Springel & Rainer Spurzem

Tutors: Lei Liu & Justus Schneider

Sheet 9 (June 16, 2011)

## 1 Heat conduction – in parallel

We consider the *time-dependent* heat condution equation in a two-dimensional periodic domain $[0, L] \times [0, L]$, and for a constant heat conductivity $\kappa$:

$$\frac{\partial T}{\partial t} = \kappa \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \tag{1}$$

We want to solve this for an intial value problem, where the initial temperature distribution is given by

$$T_0(x, y) = \begin{cases} 4T_m|x + y - L|, & \text{for } |x - L/2| < 1/8 \text{ and } |y - L/2| < 1/8, \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

Strictly speaking, a zero temperature is of course physically dubious; we here adopt it to stand for an arbitrarily small temperature.

Note that we can redefine our variables in terms of dimensionless quantitites. In particular, we may express the spatial coordinates in units of $L$, i.e. $x' = x/L$ and $y = y/L$. Likewise, we can express the temperatures in units of $T_m$, and the time as $t' = t/(L^2/\kappa)$. We can obtain the corresponding equations in these new coordinates by effectively setting $L = \kappa = T_m = 1$ in equations (1) and (2), and that's what we use in the numerical implementation.

In the present excercise, we will solve the heat conduction equation through explicit time integration, using a 2nd order Runge-Kutta scheme. Note that this is not necessarily the most efficient and most robust approach for this equation, but it is one that we can easily adapt for parallelization.

We divide the unit square in terms of $N \times N$ cells, and adopt a low-order discretization of the Laplacian with a 5-point stencil. The evolution equations for the temperatures of the cells then become a set of coupled ODEs:

$$\frac{\mathrm{d}T_{i,j}}{\mathrm{d}t} = (T_{i+1,j} + T_{i+1,j} + T_{i+1,j} + T_{i+1,j} - 4\,T_{i,j}) \tag{3}$$

- Verify that these equations are conservative, i.e. that $\sum_i T_i = const$ holds for the discretized form of the heat conduction equation.

- On the lecture's web-site (in the directory that also hold the NR routines), you can download an implementation in serial code of this time-dependent heat conduction problem, entitled `heat_serial.c`. Take a close look at this code and check that it appears to be reasonable. Run the code and write a plot script that displays the output of the code, which is the temperature profile across the diagonal of the square at a different times, overplotted in a single panel.

- Now experiment with the timestep size and verify that the integration becomes unstable for $\Delta t > 0.25\,h^2$, where $h$ is the grid spacing. Explain why this happens.
- Repeat the calculation for a couple of mesh resolutions, in particular for $N = 50$, $N = 100$, and $N = 200$. Measure the execution times. You can, for example, simply prepand the `time` command in front of the executable. How long would an execution with $N = 2000$ take?
- What is the analytic solution for $t \to \infty$?

## 2 Parallel heat conduction continued (Homework)

In this homework, you will try out three different parallelization techniques. (Note that the various code examples we consider will usually refrain from doing proper error checking on things such as memory allocation or file I/O, for clarity... In practice, you should never omit such checks!)

### 2.1 OpenMP

Copy the serial implementation into a separate file (say `heat_omp.c`), and make the following modifications to it in order to enable shared-memory parallelization on multi-core CPUs.

- Add a

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

  statement at the beginning.
- Add a

  ```
  omp_set_num_threads(2);
  printf("max threads = %d\n", omp_get_max_threads());
  ```

  at the beginning of the `main()` routine. This will set the number of threads you can use in parallel OpenMP sections to 2, and it will query this number and print it for confirmation. You can later change the number of threads by modifying the argument to the `omp_set_num_threads()` function call. (An alternative to this is to set the environment variable `OMP_NUM_THREADS` to the desired number of threads before the code is started.)
- Place a compiler directive in front of the outer loops in the routines `add()` and `get_derivative()`, of the form:

  ```
  #pragma omp parallel for private(j)
  ```

This will parallelize the corresponding loops for parallel execution.

- Compile the code with OpenMP enabled. For the gcc compiler this is done with:

  ```
  gcc -fopenmp  heat_omp.c -o heat_omp
  ```

  Note that for other compilers the syntax for enabling OpenMP can be different.

(5 pt) When implemented as described above, the OpenMP-parallelized code will have a serious bug in the function `get_derivative`. Identify this problem, and fix it. (Hint: think about the local variables in this function).

(5 pt) Then carry out some timing measurements. What speed-up can you realize for $N = 200$ when using 2 or 4 threads on a suitable multi-core machine?

## 2.2 GPU-Parallelization

We now try to parallelize the problem for graphics processing units, which nowadays feature a large number of powerful multi-stream processors. In fact, they can execute hundreds to many thousand threads in parallel, with their hardware logic being able to efficiently schedule even millions of threads.

We shall use the high-level CUDA extension of the C language to program the graphics card, because of its simplicity. This is however only available on NVIDIA hardware. The emerging OpenCL standard provides for a similar, but more portable programming interface.

In the file `head_cuda.cu`, we provide a simple adaption of the serial heat condution code to the CUDA hardware. The extension `.cu` indidates that it contains not only plain C-code, but C-code that is modified with CUDA extensions. The latter will be processed by a special C-compiler provided by NVIDIA, `nvcc`, while all the rest of the code is just standard C-code, so that CUDA-C can be viewed as a superset of standard C. For compilation of the CUDA code, you would use something like `nvcc heat_cuda.cu -o heat_cuda`, yielding the executable `heat_cuda`. Please ask your tutor in the exercise classes about access to a GPU-enabled machine. You may however also use your private PC or laptop in case it has an NVIDIA card (but you may then have to first install the CUDA developer toolkit from `http://developer.nvidia.com/cuda-toolkit-40` together with the developer version of the graphics card driver).

As you will see, in the example we procided already the following modifications/extensions of the serial code have been implemented:

- At the beginning of `main()`, the statements

  ```
  cudaGetDeviceCount(&i);
  if(i == 0)
    {
      printf("No usable CUDA device found, exiting.\n");
      exit(1);
    }
  ```

  have been added. This investigates how many CUDA capable graphics card are on the system, and if none is found, the program is stopped to avoid confusion later on.

- Next, we do not allocate our temperature field (`field[]`) and the three auxiliary fields the time integration uses (`ftemp[]`, `dfdt0[]`, `dfdt1[]`), on the host computer any more. Rather, we allocate them now in the memory of the GPU, via calls to the `cudaMalloc()` function. Correspondingly, calls to `free()` before the program ends have been converted into `cudaFree()` calls, where appropriate.

- However, the modified code still allocates memory for a separate copy of the temperature field in the array `host_field[]` on the host computer. This is done to create space for seting-up the ICs, and to hold the result from the GPU calculation before outputting it.

- After setting up the initial temperature field in `host_field[]`, we transform them to the CUDA device with a call to `cudaMemcpy()`. The actual time integration of the temperature field will then be carried out on the GPU. Because of this, the function `output_diagonal()` has been extended with a call of `cudaMemcpy()` that retrieves the result from the GPU back to the host computer.

- The most important change has been done to the functions `add()` and `get_derivative()`. They have been augmented with a `__global__` prefix, which tells the `nvcc` compiler that the corresponding functions are to be compiled as device-code which should be executed on the GPU, in a massively thread-parallel fashion. In fact, we will spawn a separate thread for each single element of our 2D-fields. In CUDA, massive armies of threads are created on the computing device by calling such device-code functions from the ordinary code. These function calls are done more or less in the standard C-fashion, except that after the function name and before the list of arguments an expression in tripple angle brackets is inserted, like this:

  ```
  function_name<<<blks,thrds>>>(arguments);
  ```

  The arguments in angle brackets specify the total number of threads that should be created and which will work on the function in parallel. The threads themselves are arranged into blocks, with each block containing a certain number of threads. The threads of a block can be indexed as a 1D list, a 2D grid or a 3D grid. Also, the blocks can be arranged as a 1D list or a 2D mesh. Which of these arrangements one uses is to a large extent a question of what is most convenient for the problem at hand (although some algorithms may also need to communicate between threads, in which case it is best to have them in the same block). We shall use blocks with $16 \times 16$ threads each, and try to cover our temperature field with a 2D-grid of blocks. The number of required blocks is then calculated as `(N+15)/16`.

  Since the functions `get_derivative()` and `add()` in principle work only on one element of the field, they do not need to use for-loops that sweep over the fields any more. However, the functions need to know on which element they are supposed to work. They are told this by CUDA-provided variables for the thread and block index of the current thread, which are stored in the variables `threadIdx.x` and `blockIdx.x`, respectively. And the same for the second dimension we use, given by `threadIdx.y` and `blockIdx.y`.

  You will also see that we have nevertheless retained a looping structure in the functions `get_derivative()` and `add()`, given by the `while`-constructs. This is done to allow the code to still work correctly if the grid of threads that is started would contain fewer elements than our $N \times N$ fields (this should however not occur in our example). In this case, we advance the location that a thread works on by the total number of threads in the x- or y-dimension, respectively, so that a thread does several elements if needed and the whole matrix is always covered.

(5 pt) Rewrite the code such that also the initial conditions are created on the GPU.

(3 pt) Why does the code use the formula `(N+15)/16` to calculate the number of required thread-blocks? What happens if $N$ cannot be divided by 16, or of $N$ is smaller than 16? Why are extra threads that are potentially launched not causing any trouble?

(2 pt) Carry out runs with $N = 50$, $N = 100$, and $N = 200$, and compare their timings to the serial code. You most likely will find a substantial speedup, somewhere in the range $\sim 5$ or so. Note that much larger speedups are possible for problems that are less dominated by memory traffic than the present example, which features very few calculations for every data elements accessed in memory.

## 2.3 Parallelization with MPI (optional material)

As a final approach to parallelization we consider splitting up the data and the computational work between several computers (or compute cores on the same machine) which run independent instances of the same code. All communication between these programms needs to be explicitely built into the algorithm.

At the lecture's web-site, there is a further version of the heat conduction code in the routine `heat_mpi.c`. To compile this code, one usually uses a command like `mpicc heat_mpi.c -o heat_mpi`, where `mpicc` is a special version of the C-compiler that automatically sets the environment correctly to allow use of the Message Passing Interface (MPI) library. An MPI-program is then started with a command of the form `mpirun -np 4 ./heat_mpi`, where in this example 4 program instances are requested. Note that depending on the system, the `mpirun` command may sometimes also be called `mpiexec`, `poe`, `orterun`, etc.

Compared to the serial code, a number of modifications have been made to enable MPI-parallelization of the code:

- An

  ```
  #include <mpi.h>
  ```

  is needed at the beginning of the code. Also, at the beginning of `main()`, the initialization of MPI is done via the calls

  ```
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &ThisTask);
  MPI_Comm_size(MPI_COMM_WORLD, &NTask);
  ```

  The variables `NTask` and `ThisTask` inform the code about the total number of MPI processes, and the rank of the current instance. These are crucial for organizing the parallel computations and the communication.

- We adopt a slab-decomposition of the temperature field, where each MPI process is taking responsibility for a certain number of rows of the matrix. This is calculated in the `Nrows` variable. The variable `FirstRow` holds the index of the first row that the current instance takes care of. A difficulty is that the total number of MPI tasks may not divide the total number of rows ($N$) evenly. In principle, one may even pick more MPI tasks than rows, such that some of them are superfluous. In order to make the code still work in this task, the variable `LastTask` stores the rank of the last MPI process that still has something to do.

- The memory allocation for the fields `field[]`, `ftemp[]`, `dfdt0[]` and `dfdt1[]` now only needs to be done for `Nrows * N` elements. This means that the data will be *distributed* in memory.

- The `set_ics()` code and the `add()` function need to be adjusted such that they only work on the small number of local rows stores on the current rank.

- The most important modification is in the function `get_derivative`. Here the problem arises that for the calculation of the derivative of the first row on any given process, access to data on another process is needed – namely the last row in the slab of the matrix that is stored on the previous processor. Similarly, this previous processor wants access to the first row on the current processor. This problem is addressed with calls of `MPI_Isend()` and `MPI_Recv()`, which exchange data about the first and last row of the current slab with the previous and next process above and below the current slab. Note that the last used MPI-process has to carefully determine the index of its last row because it may happen that it takes care of fewer than `Nrows` rows. The incoming data from the neigboring processes is stored in two temporary buffers which can be discarded after the derivative has been computed. An added difficulty is here that one may easily create deadlocks in this communication pattern, for example by using normal blocking send-calls instead of the asynchronous version `MPI_Isend()`. (Even `MPI_Sendrecv()` does not offer a simple safe solution here; this would still require to let even and odd ranks communicate in opposite order.)

- Finally, the output function `output_diagonal` also requires a few modifications. Because we want the result (the diagonal of our temperature matrix) to be stored in a single file that is most robustly written by a single process, we first need to collect the full diagonal on one of the processes. This is done by a call of the `MPI_Gather` function. However, because this collects contiguous data from each processor, and the various pieces of the diagonal we have are not in contiguous form initially, we first copy them to a small auxiliary buffer, before collecting it all on rank 0 with `MPI_Gather`. The processor with rank 0 is then writing the file with the result.

Suppose we set $N = 50$ and run the code on 15 MPI-processors. How many rows does each processor have to take care of? In the output routine, we allocate space for the diagonal of the temperature field, reserving room for `NTask * Nrows` elements. Someone suggests that it would be simpler to here allocate space for `N` elements, because this is the real length of the diagonal which is also output to file. Explain why this would introduce a bug into the code.

We may think of another clever optimization: Since `LastTask` gives the rank of the last MPI process that holds any data of our temperature field, we may restrict calling the output routine only to those ranks that hold data, for example by writing

```
if(ThisTask <= LastTask)
  output_diagonal(nr);
```

Why is this a problem? Explain the circumstances under which this will lead to a hang of the code.

Interchange the message tags 123 and 456 in the two `MPI_Recv` calls. Will the code still work?