ETH zürich

M. Troyer, P. Hadjidoukas
ETH Zentrum, ML H 44
CH-8092 Zürich

High Performance Computing for Science and Engineering I

Fall semester 2016

Set 2 - Diffusion and Multithreading

Issued: October 7, 2016

Question 1: Diffusion in 2D

Heat flow in a medium can be described by the diffusion equation of the form

$$\frac{\partial \rho(\boldsymbol{r},t)}{\partial t} = D\nabla^2\rho(\boldsymbol{r},t) \tag{1}$$

where $\rho(\mathrm{r},\mathrm{t})$ is a measure for the amount of heat at position $\boldsymbol{r}$ and time t and the diffusion coefficient D is constant. Lets define the domain Ω in two dimensions as $x, y \in [-1,1]$. We will use open boundary conditions

$$\rho(x,y,t) = 0 \quad \forall\, t \geq 0 \,\text{and}\, (x,y) \notin \Omega \tag{2}$$

and an initial density distribution

$$\rho(x,y,0) = \begin{cases} 1 & |x,y| < 1/2 \\ 0 & \text{otherwise} \end{cases} \tag{3}$$

a) Discretize equation (1) using forward Euler in time and central differences in space and write a serial code to model the time evolution of $\rho(x,y,t)$.
Hint: To run the code use the example parameters in Table 1.

Discretizing equation (1) using forward Euler yields

$$\frac{\rho_{i,j}^{(n+1)} - \rho_{i,j}^{(n)}}{\Delta t} = D\left(\frac{\rho_{i+1,j}^{(n)} - 2\rho_{i,j}^{(n)} + \rho_{i-1,j}^{(n)}}{\Delta x^2} + \frac{\rho_{i,j+1}^{(n)} - 2\rho_{i,j}^{(n)} + \rho_{i,j-1}^{(n)}}{\Delta y^2}\right), \tag{4}$$

where n is the index of the time slice $t = n \cdot \Delta t$ and i, j are the indices of the discretized spatial points $x_i = i \cdot \Delta x$, $y_i = j \cdot \Delta y$.

Serial code: `diffusion2d_serial.cpp`

b) Parallelize your code using manual C++ threads. Check that the parallel code produces the same result as the serial code and report your timings for $n = 1, 6, 12$ threads.
Hint: To run the code use the example parameters in Table 1.

A straightforward way to parallelize equation (4) on shared memory is to split the real-space domain into chunks and distribute them between different threads. For an out-of-place time step we store two copies of the density ρ and exchange their content after the synchronization point.

Parallel code: `diffusion2d_threaded.cpp`

Table 1: Example parameters.

	D	Ω	Δt
Set 1	1	128×128	0.00001
Set 2	1	256×256	0.000001
Set 3	1	1024×1024	0.00000001

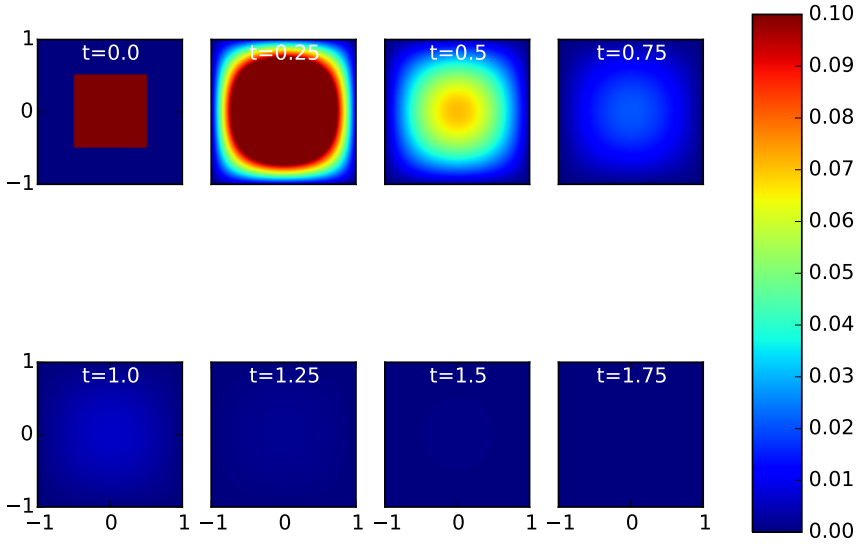

Figure 1: Density for a system using parameter set #2 (256×256 grid).

c) Make a 2D density plot of $\rho(x, y, t)$ at $t = 0, 0.5, 1, 2$.

The real-space Laplacian in equation (1) pushes the cloud away from the center and smoothens the edges of the initial density at $t = 0$, see fig. 1.

Question 2: Barrier - Synchronization with threading

A barrier is a synchronization point between multiple execution units. In this exercise we want to implement a `barrier` class using C++11 manual threading which fulfills the following syntax.

```
1 barrier b(nthreads);
2 // ... spawn `nthreads` threads ...
3 // inside each thread:
4 b.wait()
```

The `b.wait()` statement returns only when all `nthreads` called that function.

a) Implement the `barrier` class and provide a small test for it.

Code available in `barrier.hpp`, test program `barrier_test.cpp`.

In the test we synchronize 500 times all threads and we repeat the check for all possible number of threads on our system.

b) Use the barrier in the diffusion code of Question 1 such that threads are kept alive and do not respawn on each iteration. Compare the timings with your previous implementation.

In Figure 2 we plot the timings of both versions. It can be seen that for small problem sizes as the number of threads grows, the speed-up is eventually reduced to below 1. This can be understood as the time for either creation/starting the threads or keeping them synchronized via a barrier simply outweights the benefit of parallelizing the time evolution. In general, the code relying on the barrier exhibits an improved scaling over the version of the code spawning the threads every iteration. This shows how thread creation can add a significant overhead and affect overall performance.

Code: `diffusion2d_barrier.cpp`.

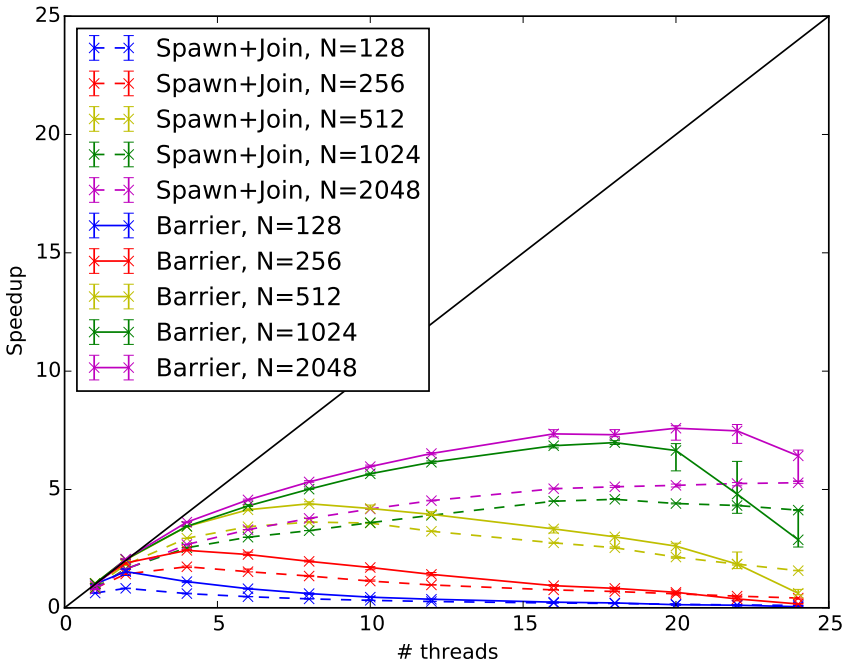

Figure 2: Strong scaling. Measured for 10'000 time-steps. Compiled with GCC 4.9 (-O3 optimization level).