Lab 3: Stencil Computing

This lab uses the heat equation as an example to explore stencil computations. We will briefly review the heat equation, its discretization, and two ways of solving the discretized equation. We will then examine code that implements the methods. We conclude with a list of possible directions for investigation.

The Heat Equation

The heat equation is a partial differential equation that models the physical transfer of heat in a region over time. Let $u(t,\vec{x},)$ represent the heat at a point $\vec{x}$ in d-dimensional space at time t, and let $u_0(\vec{x}) = u(0,\vec{x},)$ be the initial distribution of heat. In the simplest case, heat distributes over time according to the homogeneous heat equation

$$\frac{\partial u(t,\vec{x})}{\partial t} = \alpha\nabla^2 u(t,\vec{x}) \ ,$$

where α is the ***thermal diffusivity***.

In two dimensions, we have $\vec{x} = (x,y)$ and $u(t,\vec{x}) = u(t,x,y)$, in which case this equation translates to

$$\frac{\partial u(t,x,y)}{\partial t} = \alpha\left(\frac{\partial^2 u(t,x,y)}{\partial x^2} + \frac{\partial^2 u(t,x,y)}{\partial y^2}\right) \ . \tag{1}$$

A solution to the homogeneous heat equation can be used to construct solutions to a more-general inhomogeneous heat equation in which the right-hand side includes an additional $f(x,t)$ term that models heat sources or sinks.

Since the heat equation is defined with respect to continuous functions, but computers operate only on discrete values, we look for an approximate solution to the heat equation by discretizing space-time. For example, in two dimensions, suppose that we discretize the dimensions x, y, and t into points spaced Δx, Δy, and Δt apart, respectively. In the discretized space-time, each integer tuple $(n,m,\ell) \in \mathbf{Z}^3$ corresponds to a point $(t,x,y) = (n\Delta t, m\Delta x, \ell\Delta y)$ in continuous space. Ideally, we would like to determine the function u at the discrete points, i.e., determine $u(n\Delta t, m\Delta x, \ell\Delta y)$. Because of the discretization, however, we can only compute a function $U^n_{m,\ell}$ which is an approximation to the solution, that is,

$$U^n_{m,\ell} \approx u(n\Delta t, m\Delta x, \ell\Delta y) \ .$$

To compute $U^n_{m,\ell}$, one must also discretize the heat equation itself, since that equation involves continuous derivatives. Finite-difference methods are one common technique for approximating derivatives. For the heat equation, it turns out that one common approximation for the first derivative with respect to time is

$$\frac{\partial u(t,x,y)}{\partial t} \approx \frac{U^{n+1}_{m,\ell} - U^n_{m,\ell}}{\Delta t} \tag{2}$$

and a common approximation for the spatial second derivatives are

$$\frac{\partial^2 u(t,x,y)}{\partial x^2} \approx \frac{U^n_{m-1,\ell} - 2U^n_{m,\ell} + U^n_{m+1,\ell}}{\Delta x^2} \ , \quad \frac{\partial^2 u(t,x,y)}{\partial y^2} \approx \frac{U^n_{m,\ell-1} - 2U^n_{m,\ell} + U^n_{m,\ell+1}}{\Delta y^2} \tag{3}$$

```
void heat_loops(int t0, int t1)
{
  for (int t=t0; t<t1; t++)
    for (int x=1; x<M; x++)
      for (int y=1; y<L; y++)
        U(t+1,x,y) = U(t, x, y)
                   + CX * (U(t,x-1,y) + U(t,x+1,y) -2*U(t,x,y))
                   + CY * (U(t,x,y-1) + U(t,x,y+1) -2*U(t,x,y));
}
```

Figure 1: A simple loop that performs the stencil computation in Equation (4). This code calculates U on the grid for time steps n in the interval `[t0,t1)`. The loop variables `x` and `y` loop over the values of m and ℓ, respectively. In this code, `CX` $= \alpha\Delta t/\Delta x^2$, and `CY` $= \alpha\Delta t/\Delta y^2$.

Substituting the approximations in Equations (2) and (3) into Equation (1) yields the following update equation:

$$U_{m,\ell}^{n+1} = U_{m,\ell}^{n} + \frac{\alpha\Delta t}{\Delta x^2}\left(U_{m-1,\ell}^{n} + U_{m+1,\ell}^{n} - 2U_{m,\ell}^{n}\right) + \frac{\alpha\Delta t}{\Delta y^2}\left(U_{m,\ell-1}^{n} + U_{m,\ell+1}^{n} - 2U_{m,\ell}^{n}\right) . \tag{4}$$

For sufficiently small values of Δt, one can show that Equation (4) is a reasonable finite-difference scheme for solving Equation (1), that is, it is stable and accurate. For technical details, we refer the interested reader to [1].[1]

Finally, when discretizing a partial differential equation, we must specify boundary conditions for the simulation. Although $u(t,x,y)$ and $U_{m,\ell}^{n}$ are functions that are conceptually defined everywhere, we can only simulate a finite grid, that is, $0 \le m \le M$ and $0 \le \ell \le L$ for a finite time $0 \le n \le T$. For this lab, we shall assume that $u = 0$ at the boundaries. Thus, we fix $U_{0,\ell}^{n} = U_{M,\ell}^{n} = 0$ for all ℓ, and $U_{m,0}^{n} = U_{m,L}^{n} = 0$ for all m.[2]

Stencil Computations

The approximation scheme from Equation (4) falls into the general category of a ***stencil computation***. As we can see, to compute U at a point $(n+1,m,\ell)$ in step $n+1$, we require only five points from the previous time step n, namely, $(n,m\pm1,\ell\pm1)$ and (n,m,ℓ). This access pattern of points needed to calculate a value at the next time step is commonly referred to as a ***stencil***. In a stencil computation, the computation of the value at a point (m,ℓ) in space at time step n requires only "local" values, that is, values from neighboring points of (m,ℓ) from a few previous time steps. In general, one can use other approximation schemes for solving Equation (1) besides Equation (4). Many of these alternative schemes can also expressed as stencil computations. For example, one can use a higher-order approximation to derivatives, which would require a larger stencil.

Figure 1 shows pseudocode for simple nested loop which performs the stencil computation in Equation (4) for time steps n ranging between `t0` and `t1`. In this code, `U(t, x, y)` conceptually stores the values for $U_{m,\ell}^{n}$ in some abstract array data structure.

[1] This scheme for discretizing the heat equation is classified as ***forward*** in time and ***centered*** in space, since Equation (2) uses a forward difference to approximate $\partial u/\partial t$, and Equation (3) uses centered differences to approximate the $\partial^2 u/\partial x^2$ and $\partial^2 u/\partial y^2$.

[2] Another common alternative is to specify periodic boundary conditions, i.e., $U_{0,\ell}^{n} = U_{M,\ell}^{n}$ for all ℓ and $U_{m,0}^{n} = U_{m,L}^{n}$ for all m.

```
double u[2*M*L];
#define U(t, x, y) u[2*(L*(x)+(y)) + (t)&1]
```

Figure 2: A column-major layout for the array `U` and a C preprocessor macro for indexing into `U`.

The efficiency of the code in Figure 1 depends on how we store the array for `U(t, x, y)`. Since the values of `U` at a time step n only rely on values from step $n-1$, it is sufficient to store only $2ML$ values for `U`, provided that we do not care about the values of `U` at intermediate times. To exploit cache more efficiently, we may also want to store `U` in an order that matches the way we traverse the grid. For Figure 1, if we store `U` in column-major order (as shown in Figure 2), then the loop scans through `U` in order. In the case where we are interested in computing only a single time step (i.e., `t1 = t0+1`), this code is cache-efficient. There is little temporal locality to exploit because we never reuse the values computed for the final time step.

In practice, however, one is often interested in advancing the simulation by many time steps, that is, `t1-t0` is relatively large. In this case, one can better exploit temporal locality by computing `U` in an "out-of-order" fashion. More precisely, instead of computing all the values of `U` in the grid at time step n and then all values at time step $n+1$, one can advance some portions of the grid further in time than other portions.

Frigo *et al.* [3,4] give a cache-oblivious [2] algorithm for performing stencil computations. In the context of Equation (4), their algorithm operates on trapezoidal regions, where the values of `U` in the entire trapezoid can be computed, given that the values of `U` along the "bottom edges" of the trapezoid have already been computed. Their algorithm is cache-efficient because it recursively divides a trapezoid into two trapezoids of roughly equal area, and eventually the base-case trapezoids are small enough to entirely fit in cache. Figure 3 gives pseudocode for the cache-oblivious stencil computation. See the paper [3] for a thorough description of the algorithm.

Getting Started

In this lab, you will be experimenting with stencil computations using Cilk++. You can obtain the initial code using the following command:

```
git clone /afs/csail.mit.edu/proj/courses/6.884/spring10/labs/lab3/ lab3
```

This code contains two serial implementations of the stencil computation for Equation (4). The first version, in `heat_loops.cilk`, contains the looping version from Figure 1. The second version, in `heat_recursive.cilk`, contains the cache-oblivious version from Figure 3.

Building the stencil computation creates two executables, `heat.64` and `heat_demo.64`. The first executable is a command-line version that executes both the looping and recursive algorithms for the stencil computation.

The `heat.64` executable takes two arguments. The first argument M specifies the size length of the square discrete grid, and the second argument T is the final time step to compute for the simulation. For example, the command `./heat.64 400 800` discretizes space into a 400-by-400 grid of points, and solves the heat equation for 800 time steps. In the code, we have fixed $\alpha\Delta t/\Delta x^2$ and $\alpha\Delta t/\Delta y^2$ to reasonable values and started the computation with random initial values for $U^0_{m,\ell}$.

The `heat.64` executable optionally takes a `--test` argument, which runs regression tests to compare the looping and cache-oblivious versions of the code. Type `./heat.64 --test` to run these tests.

The `heat_demo.64` executable is an interactive demo that repeatedly solves the heat equation and dis-

```
const int ds = 1;
void walk2(int t0, int t1,
   int x0, int dx0, int x1, int dx1,
   int y0, int dy0, int y1, int dy1)
{
  int lt = t1 - t0;
  if (lt == 1) {
    for (int x = x0; x < x1; x++)
      for (int y = y0; y < y1; y++)
        U(t+1,x,y) = U(t, x, y)
                     + CX * (U(t,x-1,y) + U(t,x+1,y) -2*U(t,x,y))
                     + CY * (U(t,x,y-1) + U(t,x,y+1) -2*U(t,x,y));
  } else if (lt > 1) {
    if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * ds * lt) {
      int xm = (2 * (x0 + x1) + (2 * ds + dx0 + dx1) * lt) / 4;
      walk2(t0, t1, x0, dx0, xm, -ds, y0, dy0, y1, dy1);
      walk2(t0, t1, xm, -ds, x1, dx1, y0, dy0, y1, dy1);
    } else if (2 * (y1 - y0) + (dy1 - dy0) * lt >= 4 * ds * lt) {
      int ym = (2 * (y0 + y1) + (2 * ds + dy0 + dy1) * lt) / 4;
      walk2(t0, t1, x0, dx0, x1, dx1, y0, dy0, ym, -ds);
      walk2(t0, t1, x0, dx0, x1, dx1, ym, -ds, y1, dy1);
    } else {
      int halflt = lt / 2;
      walk2(t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
      walk2(t0 + halflt, t1,
    x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1,
    y0 + dy0 * halflt, dy0, y1 + dy1 * halflt, dy1);
    }
  }
}

void heat_recursive(int t0, int t1)
{
  walk2(t0, t1, 1, 0, M-1, 0, 1, 0, L-1, 0);
}
```

Figure 3: A cache-oblivious recursive method for performing the stencil computation in Equation (4). The `heat_recursive` performs the same computation as `heat_loop` in Figure 1.

plays the result on the screen.[3] After running this executable, you should see a blue window appear on your screen representing the heat-conductive material. When you move your mouse cursor into the window, the material responds as if the cursor were a heat source. This executable takes in the same arguments as the first. For example, the command `./heat_demo.64 300 150` creates a 300-by-300 grid and runs 150 time steps. This interactive version, however, repeatedly runs iterations and calculates and displays an IPS (iterations-per-second) statistic. The higher the number, the faster the stencil computation executes. By default, the demo runs the cache-oblivious algorithm, but you can choose between the looping and cache-oblivious versions using the keys `l` and `c`. The `+` and `-` keys adjust the "heat" of the cursor. Finally, `q` quits the demo.

The Fun Part

Your first goal in this lab is to speed up the stencil computation. Here are some suggestions:

- The looping version in `heat_loops.cilk` contains `for` loops that might be parallelized using `cilk_for`.
- The cache-oblivious code has a base case of `(t1-t0)==1`, leading to leaves of the divide-and-conquer tree which may be too fine-grained, so that the function-call overhead actually dominates. Try to coarsen the leaves and see what happens.
- Parallelize the cache-oblivious code in Figure 3. What is the span and parallelism of the parallel algorithm?
- The `U` array is stored using a simple column-major layout. Can you improve the performance of either the looping or cache-oblivious versions by using a different layout?

After you have managed to speed up and parallelize the stencil computation, your next goal is to explore stencil computations in some creative fashion. Some initial ideas include the following:

- Extend the algorithm to handle different stencils, such as a 9- or 25-point stencil, or a stencil that deals with the two spatial dimensions asymmetrically.
- Extend the algorithm to higher dimensions.
- Extend the algorithm to handle periodic boundary conditions.
- Find a more complicated stencil computation, such as a Lattice Boltzmann method, and parallelize it.
- Analyze whatever idea you pursue.

Remember not to turn this project into a term project unconsciously. Manage your time.

[3] In addition to the standard Cilk++ libraries, the interactive demo also requires the OpenGL library and the GLUT library to compile. These libraries are all installed on the `cagnode1` through `cagnode5`.

Exploratory Ideas

To turn this lab into a term project, here are some possible directions:

1. Find an existing application that uses a stencil computation, and speed up the application by parallelizing the code using Cilk++ and/or the cache-oblivious stencil algorithm. Since many stencil applications are written in Fortran, you may wish to investigate the Fortran/Cilk++ interface issues.

2. Extend the algorithm to handle irregularly shaped regions (e.g., 2D regions enclosed in a polygon, containing holes, etc.). How will you represent the region so as to avoid computing on points outside it? How will you efficiently determine when a point is on the boundary?

3. Some linear-algebra computations, such as LU-decomposition, can be represented as stencil computations. Develop an LU-decomposition or other code based on a divide-and-conquer stencil method.

References

[1] James F. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley-Interscience, revised edition, 2007.

[2] Matteo Frigo, Charles E. Leiserson, Harald Prokop, and Sridhar Ramachandran. Cache-oblivious algorithms. In *40th Annual Symposium on Foundations of Computer Science*, pages 285–297, New York, New York, October 17–19 1999.

[3] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *ICS '05: Proceedings of the 19th annual international conference on Supercomputing*, pages 361–366, New York, NY, USA, 2005. ACM.

[4] Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'06)*, August 2006.