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 \( \alpha \) 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 \( \Delta x, \Delta y, \) and \( \Delta t \) apart, respectively. In the discretized space-time, each integer tuple \( (n, m, \ell) \in \mathbb{Z}^3 \) 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

\[
\frac{\partial u(t, x, y)}{\partial t} \approx \frac{U_{n+1,m,\ell}^n - U_{n,m,\ell}^n}{\Delta t}, \tag{2}
\]

and a common approximation for the spatial second derivatives are

\[
\frac{\partial^2 u(t, x, y)}{\partial x^2} \approx \frac{U_{n,m-1,\ell}^n - 2U_{n,m,\ell}^n + U_{n,m+1,\ell}^n}{\Delta x^2}, \quad \frac{\partial^2 u(t, x, y)}{\partial y^2} \approx \frac{U_{n,m,\ell-1}^n - 2U_{n,m,\ell}^n + U_{n,m,\ell+1}^n}{\Delta y^2}. \tag{3}
\]
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 $\ell$, 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:

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

For sufficiently small values of $\Delta t$, 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].

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, $0 \leq m \leq M$ and $0 \leq \ell \leq L$ for a finite time $0 \leq n \leq T$. 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 $\ell$, and $U_{m,0}^{n} = U_{m,L}^{n} = 0$ for all $m$.

**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, $(n,m \pm 1, \ell \pm 1)$ and $(n,m,\ell)$. 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 $(m,\ell)$ in space at time step $n$ requires only “local” values, that is, values from neighboring points of $(m,\ell)$ 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 $\ell$ 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., \( t_1 = t_0+1 \)), 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, \( t_1-t_0 \) 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 \( U_{m_0}^{0} \).

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, x0, 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. 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.

---

*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**


Stencil Computing

Summary

In this lab, I look at stencil computations. I consider two particular implementations based on the 5 point stencil for the heat computation and then consider more general stencils.

- Section 1 discusses the parallelization for 5-point heat equation stencil. This implementation is simply based on using `cilk_for` as opposed to standard `for`.

- Section 2 discusses the parallelization of the recursive, cache-oblivious algorithm developed by Frigo which is based on trapezoids.

- Section 3 builds on the mechanical construction of the recursive algorithm by describing a stencil generator that I wrote.

1 Parallelizing Loops

It’s always good to start out with a straightforward implementation to ensure that we know what is going on, so we’re going to start by considering the simplest implementation and its simple parallelization. Just using `cilk_for` achieves pretty reasonable parallelism.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial</td>
<td>583108</td>
<td></td>
</tr>
<tr>
<td>par-both</td>
<td>100514</td>
<td>5.8</td>
</tr>
<tr>
<td>par-outer</td>
<td>92729</td>
<td>6.29</td>
</tr>
</tbody>
</table>

Figure 1: Speed ups over sequential for loop based code. Running on `cagnode2` (8 cores). Problem size: $5000 \times 5000 \times 500$.

Based on the structure of the computation, this isn’t very surprising, since there are very few data dependencies. A simple work-span analysis confirms our results are believable. Assume that the problem size is $n \times n \times t$.

$$T_1(n,t) \in \Theta(t \cdot n^2)$$
$$T_\infty(n,t) \in \Theta(t \cdot \lg^2 n)$$
$$Par(n,t) \in \Omega(\frac{n^2}{\lg n})$$

Scalability results for the loop variants are given in Figure 2. It is surprising that the burdened speedup is so high, even when both loops are parallelized, in fact, `cilkview` reports a slightly better burdened speedup when both loops are parallelized. Despite this, as we saw in the speedup results, this decreased burden does not translate to faster execution. Our runs are extremely close to perfect linear speedup which suggests that, even this very simple approach is very effective for implementing stencil computations.

2 Parallelizing Recursive, Cache-Oblivious Algorithm

The loops code does well in terms of parallelism, even when $n$ is large. However, the sequential version experiences poor cache performance. This suggests that perhaps a cache-oblivious algorithm would be able to achieve better raw performance. We look at the cache-oblivious algorithm proposed by Frigo for this. Figure 3 gives speedup results for the recursive, cache-oblivious algorithm with the following variants:
**Figure 2:** Loop scalability results, running on cagnode2 (8 cores). Problem size: $2250 \times 2250 \times 400$.

**Algorithm** | **Time (ms)** | **Speedup**
--- | --- | ---
rec-serial | 680144 | –
rec-space-space | 82696 | 8.22
rec-space-space-time | 67799 | 10.03

**Figure 3:** Speed ups over sequential for recursive, cache-oblivious algorithm. Running on cagnode2 (8 cores). Problem size: $5000 \times 5000 \times 500$.

While serial performance is considerably worse than the loop version, the speedups are much better. When we split only in space (1 or 2 dimensions) or only in time, we get an 8.22x speedup. When we allow splitting by both space and time, we get a 10.03x speedup over serial. Both of these are better than linear speedup since we are running on 8 cores, which suggests that we are benefiting from the better cache performance of the algorithm, but the cilkview results do not confirm this. The speedup from space-space to space-space-time suggests that higher dimensions using the same approach might see even more added benefits with additional processors.

To get an idea of the work-span analysis, consider an idealized case in which at every step we divide the computation into 8 perfectly equal sized pieces which corresponds to simultaneously cutting in all spacial dimensions and in time. The work is the same as above.

\[
T_1(n, t) \in \Theta(t n^2)
\]
\[
T_\infty(n, t) = 4T_\infty(\frac{t}{\sqrt{2}}, \frac{t}{\sqrt{2}}) + \Theta(1)
\]
\[
T_\omega(n, t) \in \Theta(n\sqrt{nt^2})
\]
\[
Par(n, t) \in \Omega(\sqrt{n^2t})
\]
We can use this simple analysis to consider a more detailed analysis of the span which addresses the uneveness of the sub-problems. However, again we will only consider the instances where our partition is space-space-time, with a recursive scheme of 1-3-3-1. In the equations, we will let \( d \) be the fan out of the stencil, i.e. farthest cell from the center cell where distance is measured in a single dimension. In the heat problem, \( d = 1 \), though in other stencil computations, \( d \) can be larger, for example in a 5x5 stencil, \( d = 2 \). This analysis focuses on case in which the trapezoid sides slant inward as shown in Figure 5.

\[
T_w(x,y,t) = T_w\left(\frac{x}{2} + d, \frac{y}{2} + d, \frac{t}{2}\right) + \max \left[T_w\left(\frac{x}{2} + d, \frac{y}{2} - d, \frac{t}{2}\right), T_w\left(\frac{x}{2} - d, \frac{y}{2} + d, \frac{t}{2}\right), T_w\left(\frac{x}{2} - d, \frac{y}{2} - d, \frac{t}{2}\right)\right] +
\max \left[T_w\left(\frac{x}{2}, \frac{y}{2} - d, \frac{t}{2}\right), T_w\left(\frac{x}{2}, \frac{y}{2} + d, \frac{t}{2}\right), T_w\left(\frac{x}{2}, \frac{y}{2} + d, \frac{t}{2}\right)\right]
\]

Substituting our previous “solution,” leads to the following:

\[
\sqrt[n]{\frac{3}{4}n} \quad \Theta \left( \frac{3}{2}(\sqrt{\frac{3}{4}d})(\sqrt{\frac{3}{4}d} + d) + \frac{3}{2}(\sqrt{\frac{3}{4}d} + d)\frac{t}{2} + \frac{3}{4} \frac{3}{4} + \frac{3}{4} \frac{3}{4} \right)
\]

\[
= \Theta \left( 4 \sqrt[n]{\frac{3}{4}n} \right)
\]

\[
= \Theta \left( 2 \sqrt[n]{\frac{3}{4}n} \right)
\]

\[
= \Theta \left( \sqrt[n]{\frac{3}{4}n} \right)
\]

In the first step after substituting, the only thing that I think I can do is approximate \( d \) to be a constant which therefore can be considered small compared to \( x, y, t \). This checks out, but the math seems a little sketchy to me. The analysis yields a resulting parallelism of

\[
\text{Par}(n,t) \quad \in \quad \Omega\left( \sqrt[n]{\frac{3}{4}n} \right)
\]

Unlike the loop version, the parallelism of the recursive version depends on the time. Also, note that the parallelism is considerably less using the recursive algorithm. So in the limit, we didn’t expect this algorithm to have better parallelism, only better cache-locality.

Scalability results for the parallelization of the cache-oblivious algorithm are given in Figure 4. While the empirical numbers gave super-linear scalability in both cases, cilkview only reports linear scalability on the space-space-time partitioning. The space-space only variant still performs decently, but not quite linear scaling with a much worse burdened speedup and much worse parallelism ceiling.

### 3 Generalize to Higher Dimensions

Our intuition about parallelizing the recursive, cache-oblivious algorithm is based on simply drawing a picture with the appropriate cuts. This works well in the 1-dimensional case since 2-dimensional pictures are still easy to draw (note that the second dimension here is time). Figure 5 shows a parallelizing cut strategy for 1-dimension. Essentially, we simultaneously construct a time cut and a space cut. We note that the bottom left piece (labeled 1) does not depend on the other pieces and so can be executed first. Next, the two pieces labeled 2, depend only on piece 1, so we can compute those two pieces in parallel. Finally, the upper right piece depends on both of the 2s and so should can be executed last.

In general, we can break larger dimensions in a similar way, though there is additional flexibility. In the figure on the left, we perform a space cut (the more vertical plane) and a time cut (the horizontal plane) and we can achieve the same structure, a 1-2-1. The same happens if we do two space cuts in different dimensions, as shown on the right. However, with the added dimension, we can make two space cuts and a time cut simultaneously yielding the partitioning on the third partitioning with the dependency structure 1-3-3-1. For this last case, we can assume that partitioning time always works and we can reason about spatial dimensions by looking at the top and bottom views

---

1 I’m fairly sure that this math is wrong, but I’m not entirely sure how to work it out. I feel like since all of the function arguments are the same, I can treat this as something like \( T_w(n') = 4T_w\left(\frac{n}{4}\right) + \Theta(1) \) and then plug in \( n' = n^2 \) which gives me that recurrence by the master-method.
Figure 4: Scalability analysis of the recursive cache-oblivious algorithm. Running on cagnode2 (8 cores). Problem size: 5000 × 5000 × 500.

Figure 5: Parallelizing cut for 1D.

and trying to interpolate. The last figure in Figure 6 shows the top and bottom views with some of the cruft removed. Note that taking any point in a region and drawing downward lines with slope between 1 and -1 yields only points that have already been evaluated.

Figure 6: Parallelizing cuts for 2D. (1) Space-Time (2) Space-Space (4) Space-Space-Time

Unfortunately, it isn’t immediately obvious that the dependency structure works this way and even for 2 dimensions plus time it’s difficult to visualize. An alternative way to view the parallelism is to look at code. Consider the call-tree in Figure 7 in which we partition on x-axis then on the y-axis (on the left) or on the y-axis and then on the x-axis (on the right). Note that both trees are correct based on the correctness of the 1D cuts, but have no parallelism. Now, if we think of inlining the second level calls in each of the first level calls, we get two different functions but both must be correct since we didn’t perform any transformation that changed the semantics of the program. Since both inlinings
are correct, any function calls which switch relative positions must be independent. Therefore, we can conclude that the two middle calls are indeed independent. Unlike the graphical depictions which suffer from difficulties when visualizing $n$-dimensional “trapezoids,” we can use this function inlining approach to justify the pattern of being able to perform multiple partitions simultaneously.

Based on the above observations, I wrote a python script (included in the source) which generates a generic recursive procedure for an arbitrary number of dimensions \(^2\). Figure 8 shows the code it generates for three dimensions when it only partitions on time as a last resort. An alternative is to allow cutting on time on every recursive call which produces even longer code, though, theoretically with higher parallelism.

\(^2\)I didn’t have enough time to thoroughly test the generated code or collect performance metrics on it.
Code for a 3D stencil computation.
Lab3: branch mispredictions in the cache oblivious stencil algorithm

For this lab I investigated the benefit of the cache oblivious stencil algorithm. Surprisingly I found that the cache oblivious stencil algorithm, performed slightly worse than the loop-based algorithm on a single core and up to 50% slower than the loop-based algorithm on multiple cores. The reason for lackluster performance appears in part to be from the extra branch mispredictions incurred by the cache oblivious algorithm and in the parallel case the lack of parallelism.

The benefit of the cache oblivious stencil algorithm is that it requires less fine tuning than algorithms tuned to a specific cache hierarchy, but incurs a number of caches misses on the order of the finely tuned algorithms. Some potential problems with the cache oblivious stencil algorithm are that it executes more instructions, causes more branch mispredictions, and has less parallelism than simple loop-based algorithms. For the cache oblivious stencil algorithm to outperform the loop-based algorithm the benefit of fewer cache misses must outweigh the cost of executing more instructions, incurring more branch mispredictions, and exposing less parallelism. This rest of this report analyzes this cost trade-off for the cache oblivious stencil algorithm.

Single core: Figure 1 presents a break down of the cache oblivious and loop-based stencil algorithm performance on a single core. All experiments were run on a machine with 2 GHz AMD CPUs with 2.5 Mbytes of on-chip cache accessible to one core (64 Kbytes L1, 512 Kbytes L2, and 2 Mbytes shared L3). All values were calculated by taking the minimum from five runs of heat on a 512 by 512 grid for 512 time cycles. A 512 by 512 grid requires roughly 4.5 Mbytes of memory.

The cache oblivious algorithm incurs 100x fewer cache misses than the loop-based algorithm, but the loop-based algorithm executes about 1.3x faster and incurs 200x fewer branch mispredictions. The cache oblivious algorithm must perform extra work, reflected in the 1.4 increase in instructions executed, to sub-divide trapezoids and make recursive function calls.

The cost of branch mispredictions in the cache oblivious algorithm eliminates the benefit gained from reducing cache misses. It is difficult to characterize the branch misprediction penalty precisely, but twice the pipeline depth seems to be a reasonable estimate. The AMD CPU has a 12-stage pipeline, so the estimated branch misprediction penalty is 24 cycles. The penalty of a cache miss can be as much as 300 cycles on the AMD machine, however, in practice the hardware prefetcher is able to reduce the latency to about 37 cycles when the CPU performs regular access patterns (e.g. unit strides through an array). The cost of cache misses and branch mispredictions for the cache oblivious algorithm is:

\[
\begin{align*}
\text{CO cache miss penalty} &= 37 \times 412468 = 15261316 = 1.53 \times 10^7 \text{cycles} \\
\text{CO branch mispredict penalty} &= 24 \times 56509670 = 1356232080 = 1.36 \times 10^9 \text{cycles} \\
\text{CO total penalty} &= 15261316 + 1356232080 = 1371493396 = 1.37 \times 10^9 \text{cycles}
\end{align*}
\]

and for the loop-based algorithm is:

\[
\begin{align*}
\text{LB cache miss penalty} &= 37 \times 37885154 = 1401750698 = 1.40 \times 10^9 \text{cycles} \\
\text{LB branch mispredict penalty} &= 24 \times 412468 = 9899232 = 9.90 \times 10^6 \text{cycles} \\
\text{LB total penalty} &= 1401750698 + 9899232 = 1411649930 = 1.41 \times 10^9 \text{cycles}
\end{align*}
\]

Therefore, the cache oblivious algorithm reduces cache misses on a single core, but this benefit is eliminated by the increase in branch mispredictions. Furthermore, the loop-based algorithm executes fewer instructions, and performs faster than the cache oblivious algorithm.

Multiple cores: The branch mispredict behavior and cache miss behavior of the cache oblivious stencil algorithm and the loop based algorithm is similar to the single core case. Therefore, the important factor in determining performance on multiple cores is the amount of parallelism. The parallel cache oblivious algorithm is given in Figure 3 and parallel loop-based algorithm is given in Figure 4.

\[^{1}\text{See “Characterizing the Branch Misprediction Penalty” by Eyerman et al.}\]
Figure 2 presents the performance and the parallelism measured by cilkview of the parallel cache oblivious and parallel loop-based stencil algorithm performance running on 16 cores. The loop based algorithm performs 1.5 times faster than the cache oblivious algorithm, because it has about 278 times more parallelism.

**Miscellaneous:** I would like to understand the tradeoffs of cache oblivious algorithms better. For example, whether many cache oblivious algorithms execute more instructions and trade a reduction in cache misses for an increase in branch mispredictions (or other costly performance penalties). If the behavior I identified in this lab is common, it would be interesting to analyze cache oblivious algorithms using an ideal branch misprediction model (or other ideal hardware event model), as well as the ideal cache model.

Exploring better ways to parallelize the cache oblivious stencil algorithm is also an interesting direction to explore. Parallelizing by doing more time cuts and space cuts adds extra cycles and might also increase branch mispredictions, because space cuts are not always possible. A different approach to parallelization would be interesting to explore.
Figure 1: Performance breakdown of loop-based and cache oblivious (recursive) stencil algorithms in terms of hardware events. “LLC misses” refers to last-level cache misses (i.e., L3 misses on the AMD machine). “loops relative” performance was calculated by dividing the value for recursive by the value for loops.

<table>
<thead>
<tr>
<th></th>
<th>cycles</th>
<th>instructions</th>
<th>LLC misses</th>
<th>branch mispredicts</th>
</tr>
</thead>
<tbody>
<tr>
<td>loops</td>
<td>7,979,172,160</td>
<td>4,016,358,375</td>
<td>37,885,154</td>
<td>267,717</td>
</tr>
<tr>
<td>recursive</td>
<td>10,529,957,602</td>
<td>5,624,148,336</td>
<td>412,468</td>
<td>56,509,670</td>
</tr>
<tr>
<td>loops relative</td>
<td>1.3</td>
<td>1.4</td>
<td>0.01</td>
<td>211</td>
</tr>
</tbody>
</table>

Figure 2: The performance and parallelism of the loop-based and cache oblivious (recursive) stencil algorithms. “loops relative” values were calculated by dividing the value for recursive by the value for loops.

<table>
<thead>
<tr>
<th></th>
<th>cycles</th>
<th>parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>loops</td>
<td>1,233,283,170</td>
<td>967.57</td>
</tr>
<tr>
<td>recursive</td>
<td>1,846,786,301</td>
<td>3.47</td>
</tr>
<tr>
<td>loops relative</td>
<td>1.5</td>
<td>278.83</td>
</tr>
</tbody>
</table>
void heat_recursive(SimState *Q,
    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 <= cor) {
        for (int t = t0; t < t1; t++)
            kernel_single_timestep(Q, t, x0, x1, y0, y1);
    } 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;
            int halflt = lt / 2;
            cilk_spawn walk2(Q, t0, t0 + halflt, x0, dx0, xm, -ds, y0, dy0, y1, dy1);
            walk2(Q, t0, t0 + halflt, x0, dx0, xm, -ds, y0, dy0, y1, dy1);
            cilk_sync;
            walk2(Q, t0 + halflt, t1,
                xm - ds * halflt, -ds,
                x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0,
                y1 + dy1 * halflt, dy1);
        } else if (2 * (y1 - y0) + (dy1 - dy0) * lt >= 4 * ds * lt) {
            int ym = (2 * (y0 + y1) + (2 * ds + dy0 + dy1) * lt) / 4;
            int halflt = lt / 2;
            cilk_spawn walk2(Q, t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, ym, -ds);
            walk2(Q, t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, ym, -ds);
            cilk_sync;
            walk2(Q, t0 + halflt, t1,
                x0 + dx0 * halflt, dx0,
                x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0,
                ym - ds * halflt, -ds);
            cilk_sync;
            walk2(Q, t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
            cilk_sync;
            walk2(Q, t0 + halflt, t1,
                x0 + dx0 * halflt, dx0,
                x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0,
                y1 + dy1 * halflt, dy1);
        } else {
            int halflt = lt / 2;
            walk2(Q, t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
            walk2(Q, t0 + halflt, t1,
                x0 + dx0 * halflt, dx0,
                x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0,
                y1 + dy1 * halflt, dy1);
        }
    }
}

Figure 3: A parallel version of the cache oblivious stencil algorithm. Parallelism is exposed by making one space and one time cut. cor is a coarsening factor.
void heat_loops(SimState *Q) {
    for (int t=data->t0; t<data->t1; t++) {
        for (int x=data->x0; x < data->x1; x += xcor) {
            for (int y=data->y0; y < data->y1; y += ycor) {
                kernel_single_timestep(Q, t, x, MIN(data->xl, x + xcor), y, MIN(data->yl, y + ycor));
            }
        }
    }
}
1 Parallelizing Looping Code

I first tried to parallelize the looping code. My first attempt was to put a cilk_for around the $x$-loop. I wasn’t sure if this would produce correct code with the $y$-loop inside, so I moved it to the inner loop. This resulted in a slowdown of the code. Using ./heat.64 400 800, the time went from 1,345 ms to 11,848 ms. This is nearly a $10\times$ slowdown, and obviously not the right thing to do.

I then replaced the loops to their original order ($t$, $x$, $y$) and parallelized the $x$-loop again. This resulted in a $2\times$ speedup, running in 665 ms for the same problem size. A final, reckless attempt was to parallelize the $t$-loop, which failed the correctness tests. I had gotten $2\times$ performance essentially for free; now to exceed it with parallel, cache-oblivious code.

2 Cache-oblivious Code

The first step was to coarsen the recursion. Here is my code to do that:

```c
void walk2(const SimState* Q,
    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 < THRESHOLD) {
        for (int t = t0; t < t1; t++) {
            kernel_single_timestep(Q, t, x0, x1, y0, y1);
            x0 += dx0;
            x1 += dx1;
            y0 += dy0;
            y1 += dy1;
        }
    } else if (lt > 1) {
        ...
    }
}
```
I experimented with several values for THRESHOLD, and got the following results:

<table>
<thead>
<tr>
<th>Threshold</th>
<th>Runtime (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2568</td>
</tr>
<tr>
<td>2</td>
<td>1755</td>
</tr>
<tr>
<td>4</td>
<td>1480</td>
</tr>
<tr>
<td>8</td>
<td>1429</td>
</tr>
<tr>
<td>32</td>
<td>1416</td>
</tr>
<tr>
<td>128</td>
<td>1407</td>
</tr>
<tr>
<td>512</td>
<td>1403</td>
</tr>
</tbody>
</table>

Table 1: Run-times for different coarsening thresholds.

This was getting faster, but 512 was nearly the size of the problem. And it still wasn’t competitive with the simple cilk_for implementation above. That’s because at this point, the recursive version was still serial. It’s actually surprising how much improvement there was to have just by coarsening — clearly, the cost of the recursion is very high at low levels. But even small coarsening eliminates this.

I parallelized the recursion by splitting the trapezoid in both t and x (or, equivalently, y) at each step. The bottom-left and top-right sub-trapezoids must run in sequence, but the top-left and bottom-right can run in parallel. Assuming equal size sub-trapezoids (which they will not be), this gives work and span of

\[
\begin{align*}
\text{Work}(n^2) &= O(n^2) \\
\text{Span}(n^2) &= 3 \text{Span} \left( \frac{n^2}{4} \right) + O(1) \\
&= O \left( n^2 \log_4 3 \right) \\
&= O \left( n^2 \log_2 3 \right)
\end{align*}
\]

∴ Parallelism = \frac{\text{Work}}{\text{Span}} = \frac{n^2}{n \log_2 3} \approx n^{0.415}

If we don’t assume equal-size sub-trapezoids, then we have at most an additional region of \( dt^2 / 8 \) to process, based on how we split the trapezoid. Furthermore \( dt = \Omega(n) \), so we have:

\[
\begin{align*}
\text{Span}(n^2) &\leq 3 \text{Span} \left( \frac{n^2}{4} \right) + \frac{dt^2}{2} + O(1) \\
&\leq 3 \text{Span} \left( \frac{n^2}{4} \right) + O(n^2) \\
\text{Span}(m) &\leq 3 \text{Span} \left( \frac{m}{4} \right) + O(m) \quad \text{Where } m = n^2 \\
&\leq O \left( m \log_4 3 \right)
\end{align*}
\]
\[ \text{Span}(n^2) \leq O \left( n^{21 \log_3 3} \right) = O \left( n^{\log_2 3} \right) \]

And we end up with the same asymptotic behavior.

This could be improved by splitting the trapezoid into more pieces. The code for a double split is:

```c
... } else if (lt > 1) {
    if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * ds * lt) {
        // execute two cuts -- t and x
        int xm = (2 * (x0 + x1) + (2 * ds + dx0 + dx1) * lt) / 4;
        int halflt = lt / 2;
        int th = t0 + halflt;

        walk2(Q, t0, th, x0, dx0, xm, -ds, y0, dy0, y1, dy1);
        cilk_spawn walk2(Q, t0, th, xm, -ds, x1, dx1, y0, dy0, y1, dy1);
        walk2(Q, th, t1,
                x0 + dx0 * halflt, dx0, xm - ds * halflt, -ds,
                y0 + dy0 * halflt, dy0, y1 + dy1 * halflt, dy1);
        cilk_sync;
        walk2(Q, th, t1,
                x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0, y1 + dy1 * halflt, dy1);
    } else if (2 * (y1 - y0) + (dy1 - dy0) * lt >= 4 * ds * lt) {
        // execute two cuts -- t and y
        int ym = (2 * (y0 + y1) + (2 * ds + dy0 + dy1) * lt) / 4;
        int halflt = lt / 2;
        int th = t0 + halflt;

        walk2(Q, t0, th, x0, dx0, x1, dx1, y0, dy0, ym, -ds);
        cilk_spawn walk2(Q, t0, th, x0, dx0, x1, dx1, ym, -ds, y1, dy1);
        walk2(Q, th, t1,
                x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0, ym - ds * halflt, -ds);
        cilk_sync;
        walk2(Q, th, t1,
                x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1,
                ym - ds * halflt, -ds, y1 + dy1 * halflt, dy1);
    } else {
        int halflt = lt / 2;
        walk2(Q, t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
        walk2(Q, t0 + halflt, t1,
                x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1,
                y0 + dy0 * halflt, dy0, y1 + dy1 * halflt, dy1);
    }
```
I decided to increase the problem size a bit and do a full comparison between all schemes. Results were gathered using ./heat.64 <size> <size>.

<table>
<thead>
<tr>
<th>Size</th>
<th>Loops</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Serial</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Serial</td>
</tr>
<tr>
<td>256</td>
<td>176</td>
<td>94</td>
</tr>
<tr>
<td>512</td>
<td>1.427</td>
<td>668</td>
</tr>
<tr>
<td>1024</td>
<td>18,458</td>
<td>5,394</td>
</tr>
<tr>
<td>2048</td>
<td>151,112</td>
<td>40,106</td>
</tr>
</tbody>
</table>

Table 2: Run-times for larger problem size in ms.

These results show that for large problem sizes, the recursive parallel version does best. Although it doesn’t blow the simple looping version out of the water, so it isn’t necessarily clear which version one should use if development time is a consideration (which it always is). However, limited testing with larger problem sizes shows that the cache-oblivious, recursive version gains a large advantage as the problem scales.\footnote{Specifically, with ./heat.64 4096 1024, the looped version took 78 sec and the recursive version took 42 sec.}

Also using coarsening, we are able to get the best performance from the recursive version across all problem sizes.

I also realized while running this that the coarsening threshold made a big difference – much bigger than it did in the serial case above. Using 32 or 128 gave poor performance (worse than loop), while slightly smaller values ended up being best.

3 Analysis of Parallelism as Splits Increase

Note: This analysis assumes the naive assumption that splitting at each step leaves pieces of the same size. This isn’t true for trapezoids, but it is justified by using the mutually recursive algorithm that splits into cubes and triangles at each step. This algorithm does produce regions of equal size at each step. It probably wouldn’t produce the same results for just the trapezoid case, however, because the shape of each region is significantly different.

One method to increase parallelism in this algorithm is to split $x$ and $y$ into more pieces at each iteration. This increases the fraction of the trapezoid that can be processed in parallel. My implementation above splits $x$ and $y$ in half. It is possible to split it in thirds, quarters, and so on.

When this is done, the span of the computation is essentially traversing the edge of the trapezoid. When splitting once (i.e., not splitting at all), this is a
single step. Splitting twice, this is three steps. Splitting thrice, five steps. If we
split $s$ times, then it takes $2s - 1$ sub-computations along the span.

The total work has not changed, obviously, so each subcomputation is $1/s^2$
of the original computation. In an analogy to the above equations, this gives
parallelism of:

$$
\begin{align*}
\text{Work}(n^2) &= O(n^2) \\
\text{Span}(n^2) &= (2s - 1) \text{Span} \left(\frac{n}{s}\right)^2 + O(1) \\
&= O \left( n^2 \log_s(2s-1) \right) \\
&= O \left( n^{\log_s(2s-1)} \right)
\end{align*}
$$

$\therefore$ Parallelism $= \frac{\text{Work}}{\text{Span}} = \frac{n^2}{n^{\log_s(2s-1)}}$

$= n^{2-\log_s(2s-1)}$

The parallelism asymptotically approaches $O(n)$ as $n \to \infty$, but its growth
toward this is very slow. Figure 1 shows this growth up to a 25-way split (which
is probably a bit unrealistic).

![Figure 1: Growth of parallelism as $n$ becomes large.](image)

## 4 Periodic Boundary Condition

I implemented periodic boundary conditions in the loop version of the code.
This was actually trivial to do, as I left the looping code the same and just
changed the kernel:

```c
#define DISSIPATION (0.9999)
static inline void kernel_inline(const SimState* Q, 
    int t, int x, int y)
{
    if (x == 0)
```
U(Q, t+1, x, y) = U(Q, t, Q->X-2, y);
else if (x == Q->X-1)
    U(Q, t+1, x, y) = U(Q, t, 1, y);
else if (y == 0)
    U(Q, t+1, x, y) = U(Q, t, x, Q->Y-2);
else if (y == Q->Y-1)
    U(Q, t+1, x, y) = U(Q, t, x, 1);
else
    U(Q, t+1, x, y) = Q->CX * (U(Q, t, x+1, y) - 2.0 * U(Q, t, x, y) + U(Q, t, x-1, y))
    + Q->CY * (U(Q, t, x, y+1) - 2.0 * U(Q, t, x, y) + U(Q, t, x, y-1))
    + U(Q, t, x, y) * DISSIPATION;

// add the heat
U(Q, t+1,x,y) += Q->heat_inc * Raster(Q, y, x);
}

As you can see, I also added a DISSIPATION factor to keep heat from increasing without bound.

I did not get a chance to implement the recursive version of the algorithm, although it is clear how this could be done. The problem with the current recursive algorithm is the dependence between the two cuts. Because the edges of the trapezoid “wrap-around” on the x− and y− axes, it is no longer possible to process segment 1 before segment 2 (see below).

This can be easily fixed by making two cuts, and processing the “pyramid” first.

This fixes the correctness issue but does not provide any parallelism. This can be addressed in the same way before – by making two cuts in space or time. (The numbers represent the order that computation must finish, and letters represent threads that can run in parallel.)
Another observation is that just using the old version, which is incorrect, produces visually indistinguishable results from the correct looped version of the code. For applications such as graphics or interactive entertainment, this is an example of cases where races are acceptable.
1 Introduction

This lab explores the use of stencil computations to solve the heat equation. As described in Handout 8, the heat equation is a partial differential equation that models how heat spreads through an area over some period of time. Without going too much detail into the mathematics, we can essentially calculate the heat state of a mesh of points at a point in time by examining nearby points at a previous point in time. This technique is called stencil computation. By approximating the derivatives involved, we can accurately and easily model heat transfer using the equation

$$U(t+1, x, y) = U(t, x, y) + CX \ast (U(t, x-1, y) + U(t, x+1, y) - 2 \ast U(t, x, y)) + CY \ast (U(t, x, y-1) + U(t, x, y+1) - 2 \ast U(t, x, y)).$$

This equation essentially states that the heat at one point at time $t$ is the sum of the point itself and neighboring points at time $t-1$. Two methods are explored here, one being a simple loop that iterates over all points at each time step, and the other method being a more complex recursive algorithm.

2 Stencil Computations via Parallel Loops

The simplest method of calculating the stencil is to simply iterate over all points in the mesh, performing the necessary math for each point.

```c
for (int t=data->t0; t<data->t1; t++) {
    cilk_for (int x=data->x0; x<data->x1; x++) {
        for (int y=data->y0; y<data->y1; y++) {
            kernel(Q, t, x, y);
        }
    }
}
```

The function kernel is the function responsible for updating the mesh, contained in a data structure Q. This data structure contains the mesh in an array, as well as the bounds of our mesh, x0 to x1 and y0 to y1. While the kernel function takes in a parameter time, it is important to note that the stencil computation requires information from only one timestep before $t$. Thus the size of Q is
kept to roughly twice the size of the mesh, rather than $t1 \times x1 \times y1$. As with most loop code, the layout of the data structure is important, as a poor layout can lead to excessive cache misses and thus poor performance. Both column-major and row major layouts for the array are considered for performance reasons, but the main speedup was expected from parallelizing the loops. Unfortunately points at $t + 1$ depend on the points at $t$, so parallelizing the outermost loop is not an option. However, they ONLY depend at points at $t$, so parallelizing the loops iterating through the points themselves is an option. Thus, the middle loop was made into a cilk for loop. It was not effective to make the innermost loop a cilk for loop, as the excessive number of spawns caused too much overhead.

3 Stencil Computation via Recursive Trapezoids

The second method is a cache oblivious algorithm that works on the idea that not all points in a time step need to be calculated before the points in the next step can be calculated. That is, points at $x1,y1$ do not depend on points around $x0,y0$, so in some cases we can split the work into separate regions of space and time. In the one dimensional case, one can imagine a grid of points in space and time, with the x axis being space, and the y axis being time. In this grid, it is possible to cut the spacetime grid into two regions with a line that has the slope of 1. This divides the rectangular grid into a left region and a right region. Since each point in a given period of time depends only on its neighbors, which the slope line represents, one can see that no point in the left region can possibly depend on a point in the right. By repeatedly subdividing, it is possible to get the entire region into the cache, allowing for efficient computations. It is important however to note that the right region does depend on the left region, so the order of region traversal is important. In some cases a cut as described is not possible, in which case a division along time is performed, a horizontal line, and of course the region representing an earlier time is traversed before the later time period.

While this method is cache efficient, it is not very parallelizable. Frigo notes that a trapezoidal region of space and time can be divided into alternating trapezoids, with their sides have a slope of 1 or -1. He further notes that all trapezoids with their wide ends at the bottom can be visited in parallel, as all points in of these trapezoids do not depend on points in another. The same is true for the other set of trapezoids, though the first set must be visited first. This improved algorithm allows for the parallelization of loops and the cache efficiency described earlier. A clearer description of this algorithm is in reference . A code snippet that implements this subdivision into $r$ trapezoids is as follows

```c
void walk2(const SimState* Q,
            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 && (x1-x0) >= 2 * ds * lt * r) {
    //cut x
    int l = (x1-x0)/r;
    //spawn black trapezoids
    cilk_for(int i = 0; i < r-1; i++){
        walk2(Q, t0, t1, x0 + i*l, ds, x0 + (i+1)*l, -ds, y0, dy0, y1, dy1);
    }
    cilk_spawn walk2(Q, t0, t1, x0 + (r-1)*l, ds, x1, -ds, y0, dy0, y1, dy1);
    //sync
    cilk_sync;
    //spawn black trapezoids
    cilk.spawn walk2(Q, t0, t1, x0 + (r-1)*l, ds, x1, -ds, y0, dy0, y1, dy1);
    cilk_for(int i = 1; i < r; ++i){
        walk2(Q, t0, t1, x0 + i*l, -ds, x0 + i*l, ds, y0, dy0, y1, dy1);
    }
    walk2(Q, t0, t1, x1, -ds, x1, dx1, y0, dy0, y1, dy1);
}

The walk function recursively subdivides a region of space and time into smaller regions using the method described above. Provided that the region is large enough, and that time step is greater than one, then r-1 black trapezoids are calculated in parallel, and then r+1 gray trapezoids are spawned and calculated. The base case is excluded from the snippet above, which could be set for a particular number of time steps at which point the loops algorithm could be used. Parameters to experiment with here are the the number of trapezoids generated,r, and the the base case used.

4 Analysis

Both algorithms visit each point in space and time but once to calculate it, with a constant amount of work performed at each point, so the work is $O(T \times X \times Y)$, with the T being the number of time steps, and X and Y being the dimensions of the grid. With the parallelization we must still visit all T time steps, but the spawns keep the longest time step from taking only $O(Y)$, thus making the span $O(T \times Y)$ to go through all T timesteps each taking $O(Y)$. Cilkview confirms this is pretty accurate for values of 300 for all paramters.

<table>
<thead>
<tr>
<th>Work</th>
<th>1949469089 instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Span</td>
<td>6786389 instructions</td>
</tr>
<tr>
<td>Burdened span</td>
<td>6786389 instructions</td>
</tr>
<tr>
<td>Parallelism</td>
<td>287.26</td>
</tr>
<tr>
<td>Burdened parallelism</td>
<td>287.26</td>
</tr>
</tbody>
</table>
The recursive version is a bit trickier. At each level of the recursion, we either have the base case, a split into $2 \times r$ trapezoids along a dimension $d$ if $2 \times (t_1 - t_0) \times r < d_f - d_0$ or a division into two time regions otherwise. At this time, I was unable to come up with a sufficiently satisfying analysis of the recurrence, and so present the cilkview report.

1) Parallelism Profile
   
   Work : 1890467458 instructions
   Span : 10508662 instructions
   Burdened span : 10508662 instructions
   Parallelism : 179.90
   Burdened parallelism : 179.90

   In both cases we see that parallelism is sufficiently high enough for 8 cores to expect a very good speedup.

5 Results and Optimizations

The graphs below show performance figures.

   One note of interest is that the recursive version performs at the same level as the loops serially if run on one core. The recursive version is about twice as fast as the loops version for a 400 by 400 grid for 800 timesteps.

   ![](graph.png)

   As we can see from the graphs, performance is not quite the linear speedup predicted, though the recursive version performed much better in terms of speedup as well as actual runtimes. It would be interesting to examine the recursive version with an performance analysis tool such as Vtune to see the cache miss rate and other potential issues. Despite a cache oblivious algorithm, it is possible other cache issues such as false sharing are occurring.

   Other optimizations tried were to switch from column major to order major, but this caused a slowdown of about 2x for loops and 50% for the recursive version. The $r$ factor was tinkered with, and I empirically found that a value of 25 had the best performance. This factor played a large role in runtime, as an excessively low value led to a slowdown of 8x. Oddly enough using loops for a base case of X timesteps caused errors. Even performing the loops version for 2 timesteps caused the some tests to fail. It was unclear why this was so, but in interest for time, I continued other explorations.

6 Further Work

I attempted to generalize the algorithm to higher dimensions. While algorithmically this is fairly straightforward, the actual implementation is not trivial,
Trial results for 'heat_loops'

Figure 1: Loops version Worker vs Speedup

Trial results for 'heat_recursive'

Figure 2: Recursive version Worker vs Speedup
as it requires somewhat extensive changes to the code base particularly in the indexing macros. For loops this requires nothing more than another loop. The recursive version requires another space cut to be attempted in the z dimension. Unfortunately, while implementation was seemingly successful for small grids and timesteps, or at least the simple loop versions results agreed with the more error prone recursive version, segmentation faults occurred at large timesteps and grids, so an indexing error or memory allocation error probably occurred. Due to lack of time, these issues remain outstanding.

7 Reference

Matteo Frigo and Volker Strumpen. The cache complexity of multithreaded cache oblivious algorithms
Stencils : Parallel Local Computations

1 Parallel Stencil Computation

Stencil operations seem perfect for parallel execution since they are composed of many localized computations. I implemented three parallel stencil computation algorithms for the heat equation on a 2D grid. The Stencil\textsubscript{loops} algorithm simply takes the simplest stencil computation algorithm (triple loops) and uses the cilk loop to add parallelism. I also added spawn and sync commands to a cache-oblivious recursive stencil computation, called Stencil\textsubscript{Recursive}. While theoretically sound, the base case of our Stencil\textsubscript{Recursive} algorithm was coarsened to take into account the amount of computation required in the base case in the Stencil\textsubscript{Recursive–Volume–Coarsening} algorithm. All three parallel algorithms outperformed the serial versions when tested on a 400x400 grid for 400 time steps. Table 1 shows the results of the algorithm running times. Section 3 shows the effects of the size of the kernel computation on the running time for the Stencil\textsubscript{Recursive} and Stencil\textsubscript{Recursive–Volume–Coarsening} algorithms.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stencil\textsubscript{Serial–Loops}</td>
<td>1637</td>
</tr>
<tr>
<td>Stencil\textsubscript{Serial–Recursive}</td>
<td>2505</td>
</tr>
<tr>
<td>Stencil\textsubscript{loops}</td>
<td>1589</td>
</tr>
<tr>
<td>Stencil\textsubscript{Recursive}</td>
<td>294</td>
</tr>
<tr>
<td>Stencil\textsubscript{Recursive–Volume–Coarsening}</td>
<td>-</td>
</tr>
</tbody>
</table>

Figure 1: Timing results for stencil algorithm, 400x400 grid, 400 time steps

2 The Parallel Recursive Algorithm: Stencil\textsubscript{Recursive}

The fastest algorithm is the parallel recursive cache-oblivious algorithm. The Stencil\textsubscript{Serial–Recursive} algorithm (original) tries to split one dimension at a time and recurse. The dimension to split is chosen so that the other dimensions are not “too large”. The Stencil\textsubscript{Recursive} algorithm tries to split along two dimensions at once: either the X or Y dimension, and Time. However, if both X and Y are small compared to the Time dimension, the algorithm simply splits time to make the dimensions stay even in size. Once one of X, Y and T have been split, we can now compute some of the stencil operation in parallel. In particular, Figure 2 shows how the computation can be thought of as having three levels, the second of which has a parallel component in it. Figure 4 shows the code. The Stencil\textsubscript{Recursive} algorithm chooses to split the largest of the dimensions as well as time in a single pass if it can (and computes some of these splits in parallel), otherwise the algorithm functions the same as Stencil\textsubscript{Serial–Recursive}. The base case for the recursion uses both a time threshold (how many time steps left in the computation) and a “volume” threshold which quantifies how many operations (in all three dimensions) left. Time threshold of 1 and volume threshold of 64 seemed to work best.

3 Effects of Kernel Size

I analyzed the effect of different kernel sizes on the run time of the Stencil\textsubscript{Recursive} and Stencil\textsubscript{Recursive–Volume–Coarsening} algorithms. The kernels for this test compute a value using an NxN grid of adjacent elements. Figure 3 shows the runtime of the algorithms with different kernel sizes. The surprising results show that with larger kernel sizes, the Stencil\textsubscript{Recursive}, Stencil\textsubscript{Recursive–Volume–Coarsening} and Stencil\textsubscript{loops} algorithms all seem to do pretty much the same in terms of running time. This seems to suggest that the kernel size heavily negates the speedups we saw in our initial run time analysis of the algorithms.
Figure 2: Trapezoid Splitting and Parallel Stencil Computations

Figure 3: Effects of Kernel Size on Running Time for $Stencilloops$, $Stencil_{Recursive}$ and $Stencil_{Recursive~Volume~Coarsening}$
void walk3(const SimState* Q, 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 <= TIME_THRESH || vol_est(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1) < VOL_THRESH) {
        for (int t=t0, dt=0; t<t1; t++, ++dt) {
            for (int x=x0 + dt*dx0; x< (x1 + dt*dx1); x++) {
                for (int y= y0 + dt*dy0; y< (y1 + dt*dy1); y++) {
                    kernel(Q, t, x, y);
                }
            }
        }
    } else if (lt > 1) {
        // time halving
        int halflt = lt / 2;
        int tm = t0 + halflt;
        // slopes for time
        int dx0t = dx0 * halflt;
        int dx1t = dx1 * halflt;
        int dy0t = dy0 * halflt;
        int dy1t = dy1 * halflt;
        int dst = -ds * halflt;
        if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * ds * lt) {
            int xm = (2 * (x0 + x1) + (2 * ds + dx0 + dx1) * lt) / 4;
            walk3(Q, t0, tm, x0, dx0, xm, -ds, y0, dy0, y1, dy1);
            cilk_spawn walk3(Q, t0, tm, xm, -ds, x1, dx1, y0, dy0, y1, dy1);
            cilk_sync;
            walk3(Q, tm, t1, x0 + dx0t, dx0, xm + dst, -ds, y0 + dy0t, dy0, y1 + dy1t, dy1);
        } else if (2 * (y1 - y0) + (dy1 - dy0) * lt >= 4 * ds * lt) {
            int ym = (2 * (y0 + y1) + (2 * ds + dy0 + dy1) * lt) / 4;
            walk3(Q, t0, tm, x0, dx0, x1, dx1, y0, dy0, ym, -ds);
            cilk_spawn walk3(Q, t0, tm, x0, dx0, x1, dx1, ym, -ds, y1, dy1);
            cilk_sync;
            walk3(Q, tm, t1, x0 + dx0t, dx0, x1 + dx1t, dx1, y0 + dy0t, dy0, ym + dst, -ds);
            cilk_sync;
            walk3(Q, tm, t1, x0 + dx0t, dx0, x1 + dx1t, dx1, y0 + dy0t, dy0, ym + dst, -ds);
        } else {
            walk3(Q, t0, t0 + halflt, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
            walk3(Q, t0 + halflt, t1, x0 + dx0 * halflt, dx0, x1 + dx1 * halflt, dx1, y0 + dy0 * halflt, dy0, y1 + dy1 * halflt, dy1);
        }
    }
}

Figure 4: Stencil Recursive code
Lab3 Writeup — Parallel Stencil Computations

Summary of what I did

In this lab, I tried various strategies to parallelize the stencil computation by cutting up the computation space differently. The particular stencil computations that I did for this lab has three dimensions: \( X \), \( Y \), and \( T \), where \( X \) and \( Y \) represent the two dimensions in space, and \( T \) represents the time. The heat value for a particular time step in each cell \((x,y)\) in space is computed by looking at the heat values of its neighboring cells (i.e. \((x-1,y)\), \((x+1,y)\), \((x,y-1)\), \((x,y+1)\)) and its own heat value from the previous time step.

To parallelize the stencil computation, I tried the following strategies:

1. **tiling**: Recursively cut out trapezoids in one space dimension (the larger one between \( X \) and \( Y \)), and compute the trapezoids in parallel. Once the trapezoids are computed, compute the up-side-down triangles between the trapezoids in parallel. This algorithm is similar to the one described in one of the assigned reading [1], but is extended to the two dimensional space. When the space becomes too small to cut, the algorithm then falls back on cutting the time in two halves.

2. **time-space-cut**: Recursively cut in time and one space dimension (the larger one between \( X \) and \( Y \)). The Figure 1 illustrates the cut as a projection onto the \( XT \) plane. At each level of the recursion, the space is divided into four pieces, and the numbering on the block shows the order of which the block should be computed. The blocks with the same number can be computed in parallel. When the space becomes too small to cut, the algorithm then falls back on cutting the time in two halves.

3. **space-space-cut**: Recursively cut in \( X \) and \( Y \) dimensions (like we discussed in lecture), and the blocks are computed in the ordering shown in Figure 2. Once the block numbered with 1 is computed, the blocks numbered with 2 can be computed. To see why this is the case, imagine projecting downward (from top to bottom), all cells in a block numbered with 2 are covered by either the same block or block numbered with 1. That is, to compute a cell in a block numbered with 2 in time step \( t \), we know for sure that all its neighboring cells have been computed in time step \( t - 1 \). Note that, the cut shown in Figure 2 is not the only way to cut the space. One could imagine cutting the space differently as long as the block numbered with 1 has a larger surface at the bottom than at the top.

4. **max-2**: Recursively cut in two of the three dimensions, depending on which two dimensions are the larger. This is basically an adaptive combinations of the time-space-cut algorithm, and the space-space-cut algorithm.

5. **max-3**: If all dimensions are large enough, do all three cuts, which divides the space into 8 pieces. The Figure 3 illustrates the cut. Cutting up the space allows slightly more parallelism, where one can compute the three blocks numbered with 2 in parallel once the block numbered with 1 is done. Similarly, the blocks numbered with 3 can be computed in parallel, once blocks numbered with 2 is done. If not all three dimensions are large enough, the algorithm falls back to the max-2 strategy.
Experimental Results

Among all the algorithms listed above, the tiling algorithm performs the best and have the best linear speedup. I am somewhat surprised to find that the space-space-cut algorithm performs the worst. Since besides tiling and max-3, all algorithms cut the computation space into four pieces and have same spawn patterns, I thought they would perform similarly. Another surprising thing is that, the space-space-cut algorithm seems to have the longest serial running time (i.e. when running on single worker). I am pretty sure the amount of work for all algorithms are the same (since the work is simply the number of points in the computation space). I am not sure why the serial time for the space-space-cut algorithm is consistently longer. I am wondering if it has something to do with the cache effect, but I didn’t have time to look into it.

<table>
<thead>
<tr>
<th>version</th>
<th>running time - 1500 x 1500 x 500</th>
<th>speedup on 8 proc relative to serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial recursive</td>
<td>20.85</td>
<td>n/a</td>
</tr>
<tr>
<td>tiling, 8 proc</td>
<td>2.85</td>
<td>7.32</td>
</tr>
<tr>
<td>time-space-cut, 8 proc</td>
<td>6.71</td>
<td>3.11</td>
</tr>
<tr>
<td>space-space-cut, 8 proc</td>
<td>12.95</td>
<td>1.61</td>
</tr>
<tr>
<td>max-2, 8 proc</td>
<td>7.44</td>
<td>2.80</td>
</tr>
<tr>
<td>max-3, 8 proc</td>
<td>3.71</td>
<td>5.62</td>
</tr>
</tbody>
</table>

Figure 4: The running times and relative speedup of the various parallel algorithms. The first column shows the running time of the serial recursive algorithm. All running times are shown in seconds, taking the average of five runs. The parallel algorithms are executed on eight processors.

To summarize the absolute performance results, Figure 4 shows the running time on eight workers for
all the algorithm. The first row shows the running time for the serial recursive algorithm. The time is shown in seconds, taking the average of five runs. The input size is $X = Y = 1500$, and $T = 500$.

**Figure 5:** The cilkview output of the time-space cut algorithm, running on 8 processors with input 1500x1500x500.

**Figure 6:** The cilkview output of the space-space cut algorithm, running on 8 processors with input 1500x1500x500.

**Figure 7:** The cilkview output of the max-2 cut algorithm, running on 8 processors with input 1500x1500x500.

**Figure 8:** The cilkview output of the max-3 cut algorithm, running on 8 processors with input 1500x1500x500.

The parallel algorithms that cut computations into 4 pieces did not exhibit linear speedups as the number of processors go up. According to the output from cilkview, these algorithms have parallelism flatten out at some point: 5.31 for time-space cut (shown in Figure 5), 4.05 for space-space cut (shown in Figure 6), and 6.13 for max-2 cut (shown in Figure 7). On the other hand, the max-3 (shown in Figure 8) and the tiling (shown in Figure 9) algorithms do not exhibit such behaviors under cilkview. The max-3 cut algorithm has a smaller burdened parallelism compared to the tiling algorithm, however.
Analysis

I tried to analyze the tiling algorithm by extending the analysis done in [1] with an additional dimension in space, but it was a failed attempt. In particular, the analysis on $T_1$ follows through, where $T_1(x,y,t) \leq c(\sqrt{xy} - x - y - t)$, where $x$ is the average width of the trapezoid in $X$ dimension, $y$ is the average width of the trapezoid in $Y$ dimension, and $t$ is the distance in $T$ dimension.

The span analysis is not so straightforward, however. I was able to show that $T_\infty(x,y,t) \leq 2\sigma r(2(\sqrt{xy})^{1/\lg(2(\sigma r)^{-1})}h - 1)$ for the case where one does a space cut in the tiling algorithm. I can’t prove the same thing for a time cut in the tiling algorithm, however. In particular, in order for the inequality to go through, I need to show that $((x_1 y_1) + (x_2 y_2))/2 \leq xy$, where $x_1$ and $y_1$ are the average widths of the top trapezoid, and $x_1$ and $y_1$ are the average widths of the bottom trapezoid. I think it may be possible to show that such condition holds, but I am not sure how, and also unfortunately ran out of time.

References

Lab 3 Writeup

Lab Summary

For lab 3, I explored parallelized 2d stencil computations of heat equations in Cilk++. I broke up the implementation into three steps:

- parallelize and evaluate loop implementation.
- coarsen and parallelize divide-and-conquer implementation.
- extend system to support static insulation geometries.

1 Loops

First, I parallelized the simple loop implementation:

```c
for (t=data->t0; t<data->t1; t++) {
    for (int x=data->x0; x<data->x1; x++) {
        for (int y=data->y0; y<data->y1; y++) {
            kernel(Q, t, x, y);
        }
    }
}
```

Because the time loop (line 1) cannot run in parallel, I tried parallelizing both the middle loop (line 2) and the middle loop and the inner loop (lines 2 and 3). Using a `cilk_for` for just the middle loop yielded the best results.

I tested this improvement under three scenarios: a large grid with few timesteps, a medium grid with moderate timesteps, and a small grid with many timesteps. The results are summarized in Figure 1. (1a 1600, 100), (1b 400, 400), (1c 100, 1600).

![Graphs showing speedup results for parallel loops](image)

**Figure 1:** Speedup graphs for parallel loops. 1600x1600 grid 100 steps, 400x400 grid 400 steps, and 100x100 grid 1600 steps shown left to right.

As shown in the graphs, better speedups are achieved as time decreases relative to grid size, which makes sense because the span is directly related to the amount of time steps.
2 Divide-and-Conquer

Next, I coarsened the leaves of the divide-and-conquer tree. To do this, I added a coarsen constant and changed the base case from:

```java
if (lt == 1) {
    kernel_single_timestep(Q,t0,x0,x1,y0,y1);
}
```

to:

```java
if (lt <= coarsen && lt > 0) {
    for (int i = 0; i < coarsen; i++) {
        kernel_single_timestep(Q, t0, x0, x1, y0, y1);
    }
}
```

Then, I experimented with the best coarsening constant. I found that coarsening at 5 timesteps yielded consistent speedups across several scenarios. These results are summarized in table 1.

<table>
<thead>
<tr>
<th>scenario</th>
<th>no coarsening</th>
<th>coarsening = 5</th>
<th>speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = 900\times 900, t = 300$</td>
<td>4814.2 ms</td>
<td>2753.2 ms</td>
<td>1.75</td>
</tr>
<tr>
<td>$G = 600\times 600, t = 600$</td>
<td>4029.4 ms</td>
<td>2517.8 ms</td>
<td>1.60</td>
</tr>
<tr>
<td>$G = 300\times 300, t = 900$</td>
<td>1585.6 ms</td>
<td>946.6 ms</td>
<td>1.68</td>
</tr>
</tbody>
</table>

Coarsening at 5 timesteps led to speedups from 1.6 to 1.75 across various grid sizes, $G$, and numbers of time steps, $t$.

Next, I parallelized the cache oblivious algorithm. Whenever a trapezoid was too wide in terms of $x$ or $y$, I would create 4 trapezoids, by splitting by $x$ or $y$ and by $t$. Then, I called the middle two walks in parallel. As with loops, I tested with a large grid with few time steps (\(G = 900\times 900, t = 300\)), a medium grid with moderate time steps (\(G = 600\times 600, t = 600\)), and a small grid with many time steps (\(G = 300\times 300, t = 900\)). Figure 2 shows the speedups of a large grid with few steps running on a cagnode machine with 8 cores.

![Figure 2: Speedup graph of parallel divide-and-conquer. \(G = 900\times 900, t = 300\) shows about 3.5 speedup.](image-url)
Running the large grid on 8 cores yielded nearly 3.5 speedup. Cilkview reported the work to be 18461287817 instructions and the span to be 4873641695 instructions, with a parallelism of 3.79.

3 Comparison of Divide-and-Conquer and Loops

Finally, I compared the runtimes of parallelized loops and parallelized recursion. I tested on a large grid with few time steps ($G = 900 \times 900$, $t = 300$), and a small grid with many time steps ($G = 300 \times 300$, $t = 900$).

<table>
<thead>
<tr>
<th>scenario</th>
<th>parallel loops</th>
<th>parallel divide-and-conquer</th>
</tr>
</thead>
<tbody>
<tr>
<td>$G = 900 \times 900$, $t = 300$</td>
<td>1140.4 ms</td>
<td>918.0 ms</td>
</tr>
<tr>
<td>$G = 300 \times 300$, $t = 900$</td>
<td>441.0 ms</td>
<td>429.8 ms</td>
</tr>
</tbody>
</table>

As shown, the parallel divide-and-conquer performed slightly better than parallel loops on a large grid with few time steps, but about the same on a small grid with many time steps.

4 Extensions

As an extension, I added the functionality to create simple static insulation geometries. To facilitate this, I created a bitmap $b$, that had a bit per $(x, y)$ coordinate indicating whether it was insulation or not. To test my bitmap, I set the insulation bit for all of the edge coordinates and changed the edge detection test in kernel_inline:

```c
if (x == 0 || x == Q->X-1 || y == 0 || y == Q->Y-1) U(Q, t+1, x, y) = 0.0;
```

to use my bitmap:

```c
if (B(Q, y, x) == 1) U(Q, t+1, x, y) = 0.0;
```

Next, I defined addRect and addCirc functions so that a programmer could add circular and rectangular insulation geometries to the stencil demo. Figure 3 shows the addRect method, which takes in a sim state, coordinates of the rectangle, width and height of the rectangle, and whether the rectangle should be filled or not.

```c
static inline void addRect(SimState* Q, int x, int y, int width, int height, bool fill) {
    for (int i = x; i < x+width; i++) {
        for (int j = y; j < y + height; j++) {
            if (fill) B(Q,i,j) = 1;
            else if (i == x || j == y || (i == x + width - 1) || (j == y + height - 1)) B(Q,i,j) = 1;
            else B(Q,i,j)=2;
        }
    }
}
```

Figure 3: addRect is a function that allows programmers to add rectangular insulation geometries to the stencil simulation.

As shown in Figure 3, rectangles can be placed into the geometry by setting the right insulation bits. Circles work in a similar manner. Figure 4 shows the visualization of adding both circular and rectangular insulation geometries.

Finally, to allow for more complex insulations, I added the functions remRect and remCirc. These allow the programmer to reset insulation bits over rectangular and circular objects. Figure 5 shows remRect and remCirc in combination with addRect and addCirc.

As shown in Figure 3, rectangles can be placed into the geometry by setting the right insulation bits. Circles work in a similar manner. Figure 4 shows the visualization of adding both circular and rectangular insulation geometries.
Figure 4: Visualization of static insulation geometries. From left to right: a filled rectangle, a nonfilled circle, and a filled rectangle inside of a nonfilled circle.

Figure 5: Visualization of static insulation geometries. From left to right: a filled rectangle with a smaller rectangle removed from it, a filled circle with a smaller circle removed from it.

It is important to note that static geometries do not hurt performance. Instead of checking boundary conditions on every call to \textit{kernel\_inline}, we check the value of a bit. The bitmap does, however, require $\Theta(n)$ additional space, where $n$ is the size of the grid.

Conclusions, Limitations, and Further Exploration

In this lab, I parallelized loop and divide-and-conquer implementations of stencil computation. Parallelizing loops resulted in speedups of about 2.5. Coarsening the divide-and-conquer algorithm yielded speedups of nearly 2. Parallelizing divide-and-conquer, I was able to get about 3.5 speedup on 8 cores on top of coarsening. I also extended the algorithm to allow for static insulation geometries without hurting performance.

As an extension, it may be cool to look at non-static geometries (that either transform or move over time), for which a simple bitmap would not work. It also might be interesting to look at more realistic insulation models that allow for small heat flows.
1 Overview

In this lab I parallelized a cache oblivious 2-D space stencil computation using two separate mechanisms: a space-time cut and multiple space cuts. Both approaches work on any-point stencil computation. The second mechanism is implemented by two algorithms: (1) an algorithm that tries to cut the space in 4, creating two independent sub-regions and (2) an adaptive algorithm that greedily cuts the space. The first algorithm performs well and gets a linear speedup on any input size, while the adaptive algorithm gets close to linear speedup when the time limit $T$ is smaller than $M$. Both algorithms depend on the stencil computation size and perform worse for larger stencil computations.

2 Parallel Cache Oblivious Stencil Computation

The non-parallelized cache oblivious algorithm works on well-defined general trapezoids by first making either a space or a time cut to subdivide the trapezoid into smaller regions and then recursing on these regions in the appropriate order (as described in Frigo and Strumpen’s ['05] paper; I will be using the notation consistent with the paper throughout this write-up). In order to parallelize this algorithm we need to find a way to divide the initial trapezoid into some sub-regions that don’t depend on each other and can be run in parallel.

2.1 Space-Time Cut

If instead of making a space or a time cut in the original algorithm, we make both cuts at the same time, then we can divide the trapezoid into four different regions as in Figure 1. In this case, if we compute $T_1$ first, we can run $T_2$ and $T_3$ in parallel, sync, and then run $T_4$. My first parallelization algorithm tries to make a space-time cut either in the $X$ or $Y$ dimension (if neither can be done, a regular time cut is made). The results are in the Comparisons section.

2.2 Two-Independent Cut

To get more parallelism, I tried a different approach of making multiple space cuts to divide the trapezoid. We can notice that for an arbitrary stencil where $(t + 1, x)$ depends on all points $(t, x + k)$ and $k \leq \sigma^2$, if a region is bound by vertical $+\sigma$ and $-\sigma$ slope lines and $T_0$ at the base, then this region does not need any information from its space-time neighbors and is independent. The idea is then to try cutting the trapezoid using alternating $+\sigma$ and $-\sigma$ slope lines into as many as possible independent sub-regions.
Figure 1: Space-Time Cut

Figure 2: Two-Cut

Figure 3: Adaptive-Cuts Start Cases

Figure 4: Adaptive-Cuts
Algorithm 1 Parallel_Two-Independent-Cut \((Q,t_0,t_1,x_0,\Delta x_0,x_1,\Delta x_1,y_0,\Delta y_0,y_1,\Delta y_1)\)

\[
\text{if } ((x_1 - x_0) >= 4 * \Delta x * \Delta t)\{
  // divide the base in half
  \text{int mid} = (x_1 - x_0)/2;
  \text{cilk_spawn walk2_twoparsplit}(Q,t_0,t_1,x_0,\Delta x_0,x_0+\text{mid},-\Delta x_0,y_0,\Delta y_0,y_1,\Delta y_1);
  \text{walk2_twoparsplit}(Q,t_0,t_1,x_0+\text{mid},\Delta x_0,x_1,-\Delta x_0,y_0,\Delta y_0,y_1,\Delta y_1);
  \text{cilk_sync};

  \text{cilk_spawn walk2_twoparsplit}(Q,t_0,t_1,x_0,\text{dx}_0,x_0,\Delta x_0,y_0,\Delta y_0,y_1,\Delta y_1);
  \text{cilk_spawn walk2_twoparsplit}(Q,t_0,t_1,x_0+\text{mid},-\Delta x_0,x_0+\text{mid},\Delta x_0,y_0,\Delta y_0,y_1,\Delta y_1);
  \text{walk2_twoparsplit}(Q,t_0,t_1,x_1,-\Delta x_0,x_1,\text{dx}_1,y_0,\Delta y_0,y_1,\Delta y_1);
  \text{cilk_sync};
\}
\]

My first try was to just find two independent subregions in the trapezoid. The simple way is to divide the lower base \((x_1 - x_0)\) in half and make 4 cuts as in Figure 2. If the resulting trapezoids are well-formed, then we got our 2 subregions and we can compute them first in parallel and then compute in parallel the remaining “upside-down” triangles. The results are in the Comparisons section. Code snapshot can be seen in Algorithm 1 - this only includes the code for the X-dimension (the Y-dimension is analogous).

This can be extended to look for more than 2 sub-regions at a time by trying to subdivide the base into some fixed number \(j\). However, the value of \(j\) needs to be pre-set (trying different \(j\)s in each check seems too expensive). So instead of this approach, I decided to find as many cuts as possible adaptively by using a greedy multiple space cuts algorithm. Getting this algorithm to work took a significant amount of this lab’s time, so I focus mostly on it below.

2.3 Adaptive Multiple-Space Cuts

Intuitively, in 1-D, this algorithm traverses the trapezoid in the \(X_0 \to X_1\) direction cutting away independent triangular regions using +\(\sigma\) and -\(\sigma\) slope lines. The independent regions found are recursively spawn in separate threads. After their computations finish, we re-traverse the trapezoid and compute the remaining “upside-down triangles” also in parallel (note: the rightmost resulting region can be either an independent region or an upside-down dependent region) (Figure 4).

To simplify the procedure, we always cut with -\(\sigma\) slope first. Therefore, we have 2 different starting cases (Figure 3) from which we get the start point for the triangle cutting loop. If we’re in case 1(b), we cannot make any cuts and have to do a standard time cut. Otherwise, we can spawn the computation of the leftmost region of our start point and start a +\(\sigma\) / -\(\sigma\) cutting loop. Basically, we try to either find an entire triangle of base length \(= 2 * \Delta t * \sigma\) or try to make a cut of slope +\(\sigma\). If we can only make 1 line cut, we spawn off the rightmost region and stop, otherwise, we spawn off the triangle and continue. Partial code is presented in Algorithm 2.
Algorithm 2 Parallel_Multiple-Space-Cuts($Q, t_0, t_1, x_0, dx_0, x_1, dx_1, y_0, dy_0, y_1, dy_1$)

```c
int lt = t1 - t0;
if (lt <= 10){
    kernel_basecase_timestep(Q, t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
} else { // try cutting in X-DIM
    x_start, x_curr;
    baselen = 2 * lt * ds; top_x1 = x1 + dx1 * lt;
    if(dx0 <= -ds) { // CASE 2
        x_start = x0;
    }
    else { // CASE 1
        x_start = x0 + (dx0 + ds) * lt;
    }
    if((x_start <= x1) && (((dx0 != -ds) && (x_curr != x1 || dx1 != -ds)) || ((dx0 == -ds) && (x_curr + baselen/2 < top_x1)))) // CASE 1(a)&CASE 2
        {
            if(dx0 != -ds) { // left-most region
                cilk_SPAWN walk2_adaptivesplit_spawns(Q, t0, t1, x0, dx0, x_start, -ds, y0, dy0, y1, dy1);
                x_curr = x_start;
                do{
                    if (x_curr + baselen/2 <= top_x1){
                        if (x_curr + baselen <= x1){ // full triangle
                            cilk_SPAWN walk2_adaptivesplit_spawns(Q, t0, t1, x_curr, ds, x_curr + baselen, -ds, y0, dy0, y1, dy1);
                        }
                        else if ((x_curr != x1) || (ds != dx1)) { // right-most region
                            cilk_SPAWN walk2_adaptivesplit_spawns(Q, t0, t1, x_curr, ds, x1, dx1, y0, dy0, y1, dy1);
                        }
                    }
                    x_curr += baselen;
                }while (x_curr <= x1);
                cilk_SYNC;
            }
            else if (dx0 != -ds) { // right-most region
                cilk_SPAWN walk2_adaptivesplit_spawns(Q, t0, t1, x_curr, -ds, x1, dx1, y0, dy0, y1, dy1);
                break;
            }
        }
    }
    x_curr += baselen;
}while (x_curr <= x1);
```

// upside-down triangles
x_curr = x_start;
```
```
Algorithm 3 parallel_multiple-space-cuts\((Q, t_0, t_1, x_0, dx_0, x_1, dx_1, y_0, dy_0, y_1, dy_1)\)

\[
\text{const int min\_edge = min}\left((x_1 - x_{\text{curr}}), (\text{top}\_x_1 - x_{\text{curr}} - \text{baselen}/2)\right)\; ;
\]

\[
\text{const int max\_cut} = (\text{min\_edge} \geq 0) ? (\text{min\_edge} / \text{baselen}) : -1 ;
\]

cilk\_for (int i = -1; i <= max\_cut; i++) {
  if (i == -1) {
    if (dx_0 != -ds) {
      walk2\_adaptivesplit\((Q, t_0, t_1, x_0, dx_0, x_{\text{start}}, -ds, y_0, dy_0, y_1, dy_1)\);
    }
  } else {
    int xc = x_{\text{start}} + i * \text{baselen} ;
    if (xc + baselen <= x_1) {
      walk2\_adaptivesplit\((Q, t_0, t_1, xc, ds, xc + \text{baselen}, -ds, y_0, dy_0, y_1, dy_1)\);
    } else if ((xc != x_1) || (ds != dx_1)) {
      walk2\_adaptivesplit\((Q, t_0, t_1, xc, ds, x_1, dx_1, y_0, dy_0, y_1, dy_1)\);
    }
  }
}

This partial code includes corner cases that had to be added to ensure that we actually make a cut. This code spawns off threads inside the while loop which is not as efficient as a cilk\_for loop – the actual code was restructured to use cilk\_for loops instead (the for loop goes until \(min((x_1 - x_{\text{start}}), (x_1 + dx_1 \ast \Delta t - x_{\text{start}} - 2 \ast \Delta t \ast \sigma / 2)) / 2 \ast \Delta t \ast \sigma\)). A snapshot of the cilk\_for loop computing the independent triangles is shown in Algorithm 3.

In order to move this to 2-D (or any \(n\)-dimensional space), we just need to pick which dimension to cut across. We can either check all dimensions in a row until we find the one that allows the first cut or we can estimate which dimension is more suitable for cutting - in the current implementation I just compare the widths of the trapezoids and pick the one with the largest width. If the stencil deals asymmetrically with different dimensions, we might want to pick the one with the smallest slope to get more cuts.

As expected, when \(M\) is greater than \(T\) the speedup in greater and even gets to linear for significant differences. This happens because the algorithm cannot split the trapezoids with equal \(T\) and \(M\) well, we can only make one cut when they are equal. If the trapezoid is very wide, on the other hand, we can make many more cuts. Similarly, when we vary the number of dependency points in the stencil - i.e. \(\sigma\) - the higher the slope the worse the speedup - we can fit less triangles with higher slopes into the trapezoid (Figure 5-8). An approach to help with this problem is to create a hybrid between the adaptive cutting and the space-time cut: determine the maximum number of cuts that can be made in a configuration using adaptive cuts and if this number is too small, do the time-space cut instead. A similar analysis can be applied to the 2-independent-cut algorithm, although we only try to divide it in two.
Figure 5: Adaptive $\sigma = 1$: 1024,1024,1024

Figure 6: Adaptive $\sigma = 1$: 1024,100,1024

Figure 7: Adaptive $\sigma = 1$: 1024,400,1024

Figure 8: Adaptive, $\sigma = 1$: 3000, 1000, 3000
3 Comparisons

This section contains a couple of tables and figures to compare the various parallelizations of the stencil computation: looping, space-time cut, 2-cut, and adaptive-cut. All numbers were obtained with a base case threshold of 10. The two-cut algorithm gets linear speedup and has a good running time on different input sizes. The adaptive algorithm has a good execution time and gets close to a linear speedup when \( M \) is larger than \( T \). The space-time cut (as well as parallel looping) does better on larger \( M \) and \( T \).

**Table 1:** Execution Time on 8 workers

<table>
<thead>
<tr>
<th>Input Size (M,T,L)</th>
<th>Looping</th>
<th>Space-Time-Cut</th>
<th>2-Cut</th>
<th>Adaptive-Cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>256, 256, 256</td>
<td>92ms</td>
<td>155ms</td>
<td>39ms</td>
<td>113ms</td>
</tr>
<tr>
<td>1024, 1024, 1024</td>
<td>12188ms</td>
<td>92367.04</td>
<td>2615ms</td>
<td>4824ms</td>
</tr>
<tr>
<td>2048, 1024, 2048</td>
<td>19635ms</td>
<td>27785ms</td>
<td>7556ms</td>
<td>9093ms</td>
</tr>
<tr>
<td>1024, 100, 1024</td>
<td>461ms</td>
<td>664ms</td>
<td>162ms</td>
<td>186ms</td>
</tr>
</tbody>
</table>

**Table 2:** Cilkview Parallelism

<table>
<thead>
<tr>
<th>Input Size (M,T,L)</th>
<th>Looping</th>
<th>Space-Time-Cut</th>
<th>2-Cut</th>
<th>Adaptive-Cut</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024, 1024, 1024</td>
<td>1008.29</td>
<td>2.62</td>
<td>230.14</td>
<td>22.25</td>
</tr>
<tr>
<td>1024, 100, 1024</td>
<td>1008.10</td>
<td>1.72</td>
<td>208.90</td>
<td>170.35</td>
</tr>
<tr>
<td>1024, 400, 1024</td>
<td>1008.26</td>
<td>102118.04</td>
<td>183.74</td>
<td>25.31</td>
</tr>
</tbody>
</table>

**Figure 9:** Adaptive \( \sigma = 4 \): 1024,100,1024

**Figure 10:** Adaptive \( \sigma = 6 \): 1024,100,1024
Figure 11: Looping: 1024,1024,1024

Figure 12: Space-Time-Cut: 1024,1024,1024

Figure 13: Two-Cut: 1024,1024,1024

Figure 14: Adaptive-Cut: 1024,1024,1024
Figure 15: Looping: 1024,100,1024
Figure 16: Space-Time-Cut: 1024,100,1024
Figure 17: Two-Cut: 1024,100,1024
Figure 18: Adaptive-Cut, ds = 1: 1024,100,1024
Lab 3: Stencil Computations
Karen Castelletti

In this lab, we use a discretized approximation of the heat-transfer equation [1] to calculate the heat at various timesteps in a two-dimensional grid. We use this data to power a demonstration program modeling heat transfer. The discretized approximation function which we use is shown in [2].

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

[1]

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

[2]

This approximation function may be considered a five-point stencil in that the calculation of a single point's heat value depends solely upon five data points from the prior timestep: \( n, t+1, \) and \( m+1 \). The requirement for these inputs introduces a dependence into the calculations such that a given timestep \( t+1 \) cannot be computed until its nearby values in timestep \( t \) have been.

**Looped Version**

We first run a simple triply-nested loop (over time \( t \) from \( t_0 \) to \( t_1 \), horizontal-axis \( x \) from \( x_0 \) to \( x_1 \), and vertical-axis \( y \) from \( y_0 \) to \( y_1 \)) and manually update each grid location in order by calling the given kernel function. Running this code serially over a 300-by-300 grid for 300 timesteps, the running time is 281ms [3]. By making the middle loop, over the horizontal axis, into a \texttt{cilk_for} loop, we are able to achieve a parallelism of order \( n \) and reduce the runtime over the same 300-by-300 grid for 300 timesteps to 142ms. Out of curiosity, I also tried making the innermost loop over \( y \) a \texttt{cilk_for} loop. I expected the overhead of further recursion would render the gains from additional parallelism unimpressive. In fact, the overhead overwhelmed them, and this version took 197ms on the same test case, although its parallelism was higher, at \( n^2 \).

**Recursive Version**

The second approach we try requires envisioning the grid in spacetime as a trapezoid [4] as shown, with the \( t \)-axis standing vertically and either the \( x \)- or the \( y \)-axis lying horizontally on the page. The benefit of recursion here is eventually getting down to a size where we can achieve cache locality.
2D Serial Slicing

To compute this heat equation in a recursive manner in a two dimensional grid, we employ two varieties of cuts. The first is a space cut, shown in figure [4] as the black diagonal line through the center of the trapezoid, which cuts the trapezoid into two smaller figures: a smaller trapezoid and an adjacent parallelogram. Having thus split the figure, we make two sequential recursive calls to calculate first the lower left figure's values, and then those of the upper right one.

The second type of cut we make is a time cut, shown in figure [4] as a blue horizontal line. We are again able to make two recursive calls, first for the bottom half of the figure and then for the top. This does not yet reveal how we will eventually benefit in terms of parallelism, but it already makes sense in terms of cache locality.

This version of the code of course has no parallelism, and it takes $491\text{ms}$ to run the 300-by-300 grid case. It's recurrence relation is given by $T(n^3) = 2T(n^3/2) + O(1)$.

Serial and Parallel Coarsening

Before we turn to parallelization, we first consider how coarsening this recursive algorithm may benefit us. Through a binary search of values below which to coarsen the recursive version into a looped version, we find the best performance at a value of 14 for the difference between $t_1$ and $t_0$. Making this change reduces the running time on our 300-by-300 test case from $491\text{ms}$ to $299\text{ms}$. This is using the looping code as a base case without the 	exttt{cilk_for} parallelization we utilized in the looping algorithm. When we add this parallelism back into our coarsening case, we find the performance improves again – and significantly – down to $175\text{ms}$ on eight machines.

3D Slicing and Parallelization

We now extend our algorithm to allow for parallelism in the recursive version. To accomplish this, we will leave the time cut case untouched and focus our attention toward the case of the space cut. We previously made only a single cut across the x direction with respect to time t. We now make the same sort of cut that we made before, along the x-t plane, and then we make another similar cut along the y-t plane. The result is that we now have four sections which we may consider the lower left, lower right, upper left, and upper right.

We make recursive calls using these segments, but first it is beneficial to examine their dependence. The bottom left segment clearly does not depend upon the others. Both the segment above it and to its right depend upon its completion, but not on one another. They may therefore be computer in parallel. The top right segment, however, depends on both of them, and we must therefore 	exttt{cilk_sync} the threads before computing it.

We make these calls then in the sequence lower-left, lower-right and upper-left, then upper-right. To find the span of this calculation, we begin with the base case of four units $w$ of work. With this scheme, we can accomplish this $4w$ work in $3w$ time. This $4w$ completed work now becomes the next recursive call's (from the bottom up) singular unit of work, which we shall call $1w$. Thus the span of x-depth recursive calls will be $3^k$. This depth $x$ is equal to $\log_q(n^3)$, as we make four calls with $\frac{1}{4}$ of the remaining work at each step, and the total work is of course $O(n^3)$ if we make the simplifying assumption that $x \sim y \sim t$. So the total speedup is $O(n^3) / O(n^3 \cdot \log_q^3)$. The recurrence relation of
interest to us is $T(n^3) = 3T(n^3/4) + O(1)$.

When we ran this parallelized version with a serial coarsening scheme on our test case on eight machines, we achieved a running time of 278 ms. When we also included parallel coarsening, we were able to get down to 142 ms, which is our most impressive performance result achieved.

Periodic Boundaries

Finally, as an exploration, we address a condition of our simulation thus far. We have assumed that a one-sample boundary around our array, $U$, has the value of $0.0$ all along its perimeter, as shown in [4]. A common variation in such simulations is to retain the fact that the boundaries wrap around — that the top values equal the bottom and the left equal the right — while removing the restriction that those all be set to zero.

\[ U^n_{0, \ell} = U^n_{M, \ell} = 0 \text{ for all } \ell, \text{ and } U^n_{m,0} = U^n_{m,L} = 0 \text{ for all } m. \]

To calculate the value of the grid at a given timestamp, the bottom and left-axis boundaries must initially be set to zero so that the effects on these values can propagate upward and to the right. Once we reach the rightmost and/or top lines of the grid, however, we are free to continue applying the same equation, except that our equation assumes there are values on all sides of the one we wish to calculate. To get these values for the top row, we the first meaningfully calculated value in the very bottom of the corresponding column to be “above” it and use this value in our equation. The first meaningfully calculated value will not be the value in the very bottom row, as that row is currently set as a zero-boundary, but rather the value one row up in position (col 7, row 1).

We employ a similar wraparound trick to attain values for use in the calculations for elements in the right-most column, using values from the corresponding row and the first column. Once we have set a value in either the top row or the rightmost column, we then set the corresponding value on the opposite side of the grid to the same value, satisfying the constraint that the boundaries be equal in a wraparound fashion.

Adding this functionality allows the demo to show the effects of heat wrapping around the boundaries of our grid. It does not meaningfully change the algorithmic runtime of our code, although there is some additional overhead which we were previously spared by leaving effectively $4n$ elements of each $n$-by-$n$ grid uncalkulated. While our parallelism is not affected, we do observe a modest slowdown in the running time achieved on our 300-by-300 grid test case. Our running time is now back up to 174 ms.

[3] Collected Data

<table>
<thead>
<tr>
<th>algorithm</th>
<th>periodic boundaries</th>
<th>coarsening</th>
<th>time</th>
<th>parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>serial loops</td>
<td>no</td>
<td>none</td>
<td>281ms</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>parallel middle loop</td>
<td>no</td>
<td>none</td>
<td>142ms</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>parallel inner loops</td>
<td>no</td>
<td>none</td>
<td>197ms</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>serial recursive</td>
<td>no</td>
<td>14, serial</td>
<td>491ms</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>parallel recursive</td>
<td>no</td>
<td>14, serial</td>
<td>299ms</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>serial recursive</td>
<td>no</td>
<td>14, parallel</td>
<td>175ms</td>
<td>$O(n^3)/O(n^3*\log_4 3)$</td>
</tr>
<tr>
<td>parallel recursive</td>
<td>no</td>
<td>14, parallel</td>
<td>142ms</td>
<td>$O(n^3)/O(n^3*\log_4 3)$</td>
</tr>
<tr>
<td>parallel recursive</td>
<td>yes</td>
<td>14, parallel</td>
<td>176ms</td>
<td>$O(n^3)/O(n^3*\log_4 3)$</td>
</tr>
</tbody>
</table>

* All data in this table from algorithms run on a 300-by-300 grid for 300 timesteps on 8 machines.
Kevin Kelley — Lab 3

In this lab exercise, we explored methods of parallelizing stencil computations and ways of improving their serial performance. Stencil computations are often a useful way of solving discretized differential equations; in this case, we utilized a kernel modeling heat transfer in two dimensions.

I began by implementing several of the improvements suggested by the laboratory assignment. Unfortunately, I was unable to get my intended extension to those ideas (which is discussed in more detail below) to work within a timeframe approximating the one suggested.

*Performance information was generated on a Core i7 system with four physical cores and two-way hyperthreading.*

**Parallelizing heat_loops**

The more straightforward of the two algorithms we examined simply iteratively applied the kernel.

```c
for( int t = 0; t < timesteps; ++t )
    for( int x = 0; t < width; ++x )
        for( int y = 0; t < height; ++y )
            apply_kernel( t, x, y );
```

As should be obvious, the outer loop cannot be parallelized; certain elements of a time step \( t \) are required to compute the next time step \( t + 1 \). Either of the inner loops can be; however, as I expected, the best solution is to parallelize only the middle one. Parallelizing the inner one simply adds too much overhead to be practical, even though it does expose additional parallelism.

<table>
<thead>
<tr>
<th></th>
<th>Time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Serial</td>
<td>282</td>
</tr>
<tr>
<td>Inner loop only</td>
<td>2283</td>
</tr>
<tr>
<td>Middle, inner loops</td>
<td>195</td>
</tr>
<tr>
<td>Middle loop only</td>
<td>142</td>
</tr>
</tbody>
</table>

**Improvements to heat_recursive**

There were several clear directions for improvement to the original implementation of the cache-oblivious version, *heat_recursive*. I began by coarsening the leaves of the computation. This required some attention to detail, because the iteration space is subdivided into trapezoids instead of into quadrilaterals, which are easier to reason about. However, the implementation itself is quite easy; we may, at any point, stop subdividing the iteration...
space and simply apply the kernel serially. As simple as this is, doing so effects a substantial performance improvement because it substantially reduces overhead, as the laboratory assignment suggests it might.

Then, I parallelized the computation. Since each of the three cases beyond the base case simply subdivides the iteration space in half, the obvious method for parallelizing heat_recursive is to perform both of those recursive calls in parallel. Unfortunately, this is not possible; in each case, one of the subdivisions depends on the other.

This is is a consequence of the shape of the stencil in space-time. Since the value of a particular cell at time \( t \) depends on the value of neighboring cells at time \( t - 1 \), the shape on the lower/left side of the division must be computed first. Thus, in order to exploit any of the parallelism present in the problem, we must utilize a more complex system of subdivision. In particular, we must create two or more shapes such that one does not “overhang” the other. To meet this restriction, I chose to create an inverted triangle in the middle of the trapezoid; this leaves two shapes that can be recursed on in parallel, and the triangle, which cannot be handled in parallel with either of the two shapes.

|                      |    |
|----------------------|    |
| Original             | 489ms |
| Coarsened leaves     | 301ms |
| Parallelized         | 97ms  |

**An attempted extension**

One of the first things that I noticed about the suggested parallelization of the cache-oblivious algorithm is that its span is \( O\left(\frac{2}{3}n\right) \), assuming that the three subdivisions are of equal size. I also noticed that the performance improvement I saw from parallelization was only approximately by a factor of 3.1. With that in mind, I thought that perhaps decreasing the span of the computation might yield improved performance.

The implementation which is provided is also hardwired to support a particular stencil pattern in two dimensions (and therefore a three dimensional iteration space). I sought to remedy this situation by adding support for additional types of stencils.

I used a preprocessing stage to identify data dependency patterns in the iteration space of the computation (e.g., the physical space being simulated and the additional time dimension). In the most general case, each cell depends on the entire previous timestep, and so the best we can do is the solution implemented in heat_loops.

However, in some cases, such as the one presented in the laboratory assignment, more complex patterns are possible. The pattern presented has both left-facing and right-facing data dependencies in both the \( x \) and \( y \) dimensions; the consequences of this on parallelizing a traversal of this iteration space are described above.

It stands to reason that, for each combination of data dependency patterns that a stencil might have, there exists some method of subdivision and recursion that yields acceptable
parallel performance. However, my attempted implementation failed to be correct. I reasoned through what it was doing and investing a significant amount of time in debugging it, but was unable to produce something without races (and thus correctness issues) after investing more than the specified amount of time. The concept, however, is interesting and sound—and has a fundamental place in an active research project of mine.
<table>
<thead>
<tr>
<th>Method</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial loops</td>
<td>290ms</td>
</tr>
<tr>
<td>Parallel inner y for</td>
<td>2300ms</td>
</tr>
<tr>
<td>Both parallel x and y</td>
<td>200ms</td>
</tr>
<tr>
<td>Parallel outer x for</td>
<td>150ms</td>
</tr>
</tbody>
</table>

Table 1: Times taken for loop stencil implementation tests with various parallelizations

Matthew Goldstein  
6.884 Lab 3

I chose to do several of the suggested optimizations in this lab, cutting times by half for the loops version and 2/5 for the recursive implementation. That said, the looping version at its initial speed (and with all but one variant) was faster by far than the recursive. Tests were done on the default 300x300 size.

1 Initial Optimizations

This section describes optimizations in the first section of suggestions. I parallelized the loops in the looping implemention, coarsened the leaves of the recursive implementation, parallelized the leaves of the recursive implementation, and attempted a row rather than column-major array layout.

1.1 Parallelized Loops

I changed the for loops of the looping implementation to cilk_for loops in order to parallelize it. The results can be seen in Table 1. As might be expected, parallelizing the inner loop only just doesn’t make sense, while parallelizing the outer x loop cut times to nearly 1/2.

1.2 Recursion Granularity

I coarsened the base case of the recursive implementation. The results can be seen in Figure 1. Past a granularity of 5, the times did not seem to improve noticeably.

1.3 Layout and Leaf parallelizing

I switched the layout from column-major to row major. This only made run times slightly slower, which is reasonable to expect given the similarity. Switching x and y between outer and inner loops brought the times back down to where they were before, so row vs. column ordering seems to have no effect.

I also parallelized the leaves of the recursive version, switching the for loops to cilk_for loops to see if this would help. It did not, giving similar results of a factor of 10 slower as when I attempted to parallelize the inner loop of the looping implementation.

2 Other Stencil Exploration

For further exploration, I changed the stencil to deal with a 9-way stencil rather than 5. This was fairly straightforward, only requiring a change to the kernel’s math - no changes in bounds were needed though one would have been needed for a 25-point stencil. Given an algorithm for sweeting across an area, it makes sense that changing the size and shape of the stencil itself should be fairly easy.
Increasing the stencil size only increased the running time of the loops implementation from 150ms to 160ms. The recursive version, on the other hand, increased from 305ms to 415ms. In case this was due to cache size, I changed the granularity of the base case, but this did not improve the running time.

I also tried to improve the looping implementation by coarsening its inner loop somewhat. This was a mistake given that it was the very innermost loop, and running times increased.

Figure 1: Times taken by the recursive stencil implementation as leaf granularity was coarsened
1 Introduction

For this lab, I explored ways to parallelize the process of calculating heat diffusion over an x-y grid over time. In order to do this, I explored ways to subdivide the x-t or y-t space into sub-pieces that could be calculated in parallel. As discussed in class, I used the base shape of a trapezoid in an x-t space, in which every point (above the bottom row) can be calculated using the points below it in the trapezoid.

Additionally, I attempted to expand the capability algorithm to the more general case of an n-dimensional grid.

2 Parallelizing Looping Version

To parallelize the looping version of the heat diffusion calculation code I tried two alternatives. First, I parallelized just the x loop, which produced a practical speedup of a little over 3. Then, I tried parallelizing both the x and y loops. Although this ran faster than the un-parallelized version of the code, it was slower than just parallelizing the x outer loop. This suggests that the overhead of generating additional threads for processing each point in the x-y plane is more costly than the parallelizing effect. The time outputs for this are shown below.

No parallelization:
Running on 8 workers
consumed time: 503ms for heat_loops

Parallelizing X for loop:
Running on 8 workers
consumed time: 153ms for heat_loops

Parallelizing both cilk_for loops:
Running on 8 workers
consumed time: 207ms for heat_loops

If we assume that $\Theta(t) \approx \Theta(x) \approx \Theta(y)$, the work done by this algorithm is $\Theta(n^3)$. The span would be $\Theta(n^2)$, given that all n values in the x-dimension could be calculated in parallel. This produces a theoretical speedup of $\Theta(n)$. 

3 Coarsening Recursive Version

To improve the runtime of the serial recursive implementation of the heat diffusion calculation code, I coarsened it by having it perform the calculations for subspaces with $\Delta t \leq 15$ using the looping version seen in the heat_loops file. This coarsening produced significant speedup ($\sim \times 2$). After playing around with various coarsening values, I determined that a $\Delta t = 15$ cutoff seemed to produce the shortest execution time for the recursive computation. Interestingly enough, parallelizing the coarsening looping code significantly increased the execution time, presumably because the overhead of parallelizing a base case of size $\Delta t = 15$ exceeded any speedup that this would create. Below are the observed runtimes for these coarsening scenarios:

No Coarsening

Running on 8 workers
consumed time: 521ms for heat_recursive

Coarsening ($\Delta t = 15$)

Running on 8 workers
consumed time: 302ms for heat_recursive

Coarsening ($\Delta t = 15$) with parallelization

Running on 8 workers
consumed time: 1009ms for heat_recursive

4 Parallelizing Recursive Version

To parallelize the recursive version of the heat diffusion calculation code I tried two different slicing schemes. For both schemes, I did a set of simultaneous space and time cuts to split a trapezoid computation area into sub-pieces, some of which could then be computed in parallel. Figures 1 and 2 shows the two slicing schemes for subdividing a x-t space (or y-t space) into subspaces.

4.1 Four Sub-Piece Slicing

In the first scheme, I did the time and space cut simultaneously in the x and t dimensions (or the y and t dimensions) to split the trapezoid into 4 sub-pieces. I then performed the sub-computations by doing the bottom left piece first, then the next two (top left and bottom right) in parallel and finally the top right piece last. This split and runtime order is shown in Figure 1.

This means that if we are doing some $N$ amount of work to compute the x-y-t space of heat diffusion calculations, the span of the computation using this 4 piece slicing method would be:

$$T(N) = 3T(N/4) + \Theta(1)$$

(1)
Because at each level we are performing 1 computation, then 2 computations in parallel and then 1 computation, the time span is the time it takes to do 3 sub-calculations of size $n/4$. If we assume that we are doing calculations in an $n$-by-$n$-by-$n$ $x$-$y$-$t$ space, the amount of work we’re doing is $N = \Theta(n^3)$. This translates to a span of $\Theta(n^{3\log_4 3}) \approx \Theta(n^{2.377})$. Therefore, the total speedup would be $\Theta(n^3)/\Theta(n^{3\log_4 3}) \approx \Theta(n^{0.623})$.

In reality, this analysis assumes that $\Theta(t) \approx \Theta(x) \approx \Theta(y)$, which may not be the case if $t$ is significantly larger than $x$ and $y$. My parallelized algorithm handles this case by checking whether $t$ is much bigger than $x$ and $y$, and if so, performing a simple time cut. This reduces the speedup from the theoretical value calculated above, as $t$ gets larger.

Comparing the parallelized loops and recursion 4 sub-piece implementations using cilkview produced the following results:

**Statistics for heat_loops**
Parallelism : 287.26
Speedup Estimate
  2 procs: 1.99 - 2.00
  4 procs: 3.93 - 4.00
  8 procs: 7.68 - 8.00
  16 procs: 14.70 - 16.00
  32 procs: 27.04 - 32.00

**Statistics for heat_recursive**
Parallelism : 175.24
2) Speedup Estimate
  2 procs: 1.98 - 2.00
  4 procs: 3.89 - 4.00
  8 procs: 7.49 - 8.00
  16 procs: 13.97 - 16.00
  32 procs: 24.60 - 32.00

Although the parallelized loops algorithm exhibits more parallelism, in practice the recursion implementation had a faster runtime than the parallelized loops implementation:

Original Unparallelized Recursion:
Running on 8 workers
consumed time: 197ms for heat_loops
consumed time: 303ms for heat_recursive

4-Subpiece Slicing Recursion:

Running on 8 workers
consumed time: 195ms for heat_loops
consumed time: 169ms for heat_recursive

4.2 Nine Sub-Piece Slicing

For my second attempt at parallelization, I did 4 simultaneous cuts - 2 space cuts and 2 time cuts - to split the x-t (or y-t) space into 9 subspaces. This allowed me to do the calculations for these 9 subspaces in five passes, as shown in Figure 2. The 2nd and 4th passes run 2 of the sub-computations in parallel and the 3rd pass runs 3 sub-computations in parallel.

This means that if we are doing N amount of work to perform all calculations in the x-y-t space, the span of this computation can be calculated as follows:

\[ T(N) = 5T(N/9) + \Theta(1) \]  \hspace{1cm} (3)

\[ T(N) = \Theta(N^{\log_5 9}) \]  \hspace{1cm} (4)

Because at each level we are performing 9 computations in 5 passes, the time span at each level is the time it takes to perform 5 sub-computations of size n/9. If we again assume that \( \Theta(t) \approx \Theta(x) \approx \Theta(y) \), the amount of work we’re doing is \( N = \Theta(n^3) \). This translates to a span of \( \Theta(n^{3\log_5 9}) \approx \Theta(n^{1.9746}) \). Therefore, the total speedup would be \( \Theta(n^3)/\Theta(n^{3\log_5 9}) \approx \Theta(n^{0.8025}) \).

Comparing the runtime of this slicing implementation to the parallelized looping implementation in cilkview produced the following output:

Statistics for heat_loops
Parallelism : 287.26
2) Speedup Estimate
   2 procs: 1.99 - 2.00
   4 procs: 3.93 - 4.00
   8 procs: 7.68 - 8.00
   16 procs: 14.70 - 16.00
   32 procs: 27.04 - 32.00

Statistics for heat_recursive
Parallelism : 237.50
2) Speedup Estimate
   2 procs: 1.99 - 2.00
Although the parallelism and theoretical speedup for this recursive implementation are still not as good as that of the parallelized looping algorithm, they are better than the original 4-subpiece slicing scheme.

In practice, this parallelized recursive implementation ran faster than both the 4-subpiece recursive implementation and the parallelized looping implementation:

Original Unparallelized Recursion:

Running on 8 workers
consumed time: 197ms for heat_loops
consumed time: 303ms for heat_recursive

9-Subpiece Slicing Recursion:

Running on 8 workers
consumed time: 195ms for heat_loops
consumed time: 156ms for heat_recursive

5 Comparing the Two Slicing Schemes

Overall, both slicing schemes parallelized the algorithm and helped it run faster than the unparallelized recursion and faster than the parallelized loops implementation. The 9-subpiece slicing scheme was slightly better than the 4-subpiece slicing scheme, but the difference in the theoretical and practical amount of speedup between the two was not too big (∼ Θ(n^{0.1795}); 13 ms)

6 Multidimensional Heat Diffusion Calculation

For my exploration of the heat diffusion calculation, I attempted to expand the algorithm to work for any number of dimensions. Below is my code for the recursive case of the algorithm, as shown in the "Cache Oblivious Stencil Computation" article by Matteo Frigo and Volker Strumpen.

```c
else if (lt > 1)
{
    C *p;

    /* for all dimensions, try to cut space */
```
for (p = c; p < c + n; ++p) {
    int x0 = p->x0, x1 = p->x1, dx0 = p->dx0, dx1 = p->dx1;
    if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * ds * lt) {
        /* space cut */
        C save = *p;
        int xm = (2 * (x0 + x1) + (2 * ds + dx0 + dx1) * lt) / 4;
        *p = (C){ x0, dx0, xm, -ds };
        walk(Q, t0, t1, c);
        *p = (C){ xm, -ds, x1, dx1 };
        walk(Q, t0, t1, c);
        *p = save;
        return;
    }
}
{ /* time cut */
    int s = lt / 2;
    C newc[n];
    int i;
    walk(Q, t0, t0 + s, c);
    for (i = 0; i < n; ++i) {
        newc[i] = (C){ c[i].x0 + c[i].dx0 * s, c[i].dx0,
                        c[i].x1 + c[i].dx1 * s, c[i].dx1 };
    }
    walk(Q, t0 + s, t1, newc);
}

This code works in the same way that the original x-y-t space code, in that for each dimension, it goes through and checks whether the width of the trapezoid in this dimension is bigger than the height (Δt). If it is, it does the appropriate space cut. Using this code together with the original base case when there are 2 space dimensions - as in the x-y-t space implementation - I was able to test that this code worked correctly.

Adapting the base case of the algorithm proved to be more challenging, because of the required changes to the math logic of the kernel. Below is the code I would use for the base case of Δt = 1, assuming the existence of a m_kernel_single_timestep method, which would do the calculations necessary to determine the heat of one point in n-dimensions:

if (lt == 1)
{
    C *p;
    int all_x0[n];
    int all_x1[n];
    for(p=c, curr_dim = 0; p < c + n; ++p, curr_dim++)

{ int x0 = p->x0, x1 = p->x1;
  all_x0[curr_dim] = x0;
  all_x1[curr_dim] = x1;
}
m_kernel_single_timestep(Q, t0, all_x0, all_x1);
1 Looping Algorithm

Parallelizing the loops method is fairly straight forward. Because each point \((x,y,t)\) only depends on points in the plane \(T = t - 1\), we can `cilk_for` both \(x\) and the \(y\) loops.

1.1 Parallelism Analysis and Results

For a one dimensional stencil computation, this parallelized loops approach will have a work of \(O(n^3)\) if \(t, x,\) and \(y\) are on the order of \(n\). The span will be \(O(n)\) – since each \(t\) iteration completes in \(O(1)\) time when with infinite workers. This gives us a parallelism of \(O(n^2)\), which is a lot of parallelism.

`cilkview` returns a parallelism of 144621.83 for an input of \(1500 \times 1000 \times 1500\), which is more than enough to saturate the 8 workers we are running on.

2 Recursive Cache Oblivious Algorithm

The recursive cache oblivious algorithm is a bit harder to parallelize. To do this, we cut space and time simultaneously, giving us four cuts, then we can execute two of them in parallel. To do this, we must have enough width to perform the space cut or else we can only do time cuts. Figure 1 shows the pseudocode for this cut and spawn.

```java
if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * ds * lt) {
    walk2(... SOUTHWEST QUARTER ...);
    cilk_spawn walk2(... NORTHWEST QUARTER ...);
    walk2(... SOUTHEAST QUARTER ...);
    cilk_sync;
    walk2(... NORTHEAST QUARTER ...);
}
```

Figure 1: Pseudocode for the space-time cut and condition

2.1 Coarsening Coefficient

To determine the best \(t\)-step to begin coarsening at, I ran a binary search on the coarsening coefficient for inputs of \(X, Y = 1000\) and \(T = 1000\). I chose a coarsening level of 13 based on the results in Figure 2.
2.2 Parallelism Analysis

To conduct a parallelism analysis, I will begin by assuming that the cut segments are equal in size, and then loosen that assumption. Further, I will assume that space/time cuts preserve the ratio of width to height in the trapezoid. If the segments are equal, then for $N$ as the area of the traversal space, here $x \cdot y \cdot t$, the recurrence for the span $S$ is,

$$S(N) = 3S\left(\frac{N}{4}\right) + O(1)$$

Using the Master Method, $S(N) = O(N^{\log_21.732})$ (base 2), and when $x, y, t$ are on the order of $n$, $S(n) = O(n^{2.377})$ and work $W(n) = O(n^3)$. So, parallelism is approximately $O(n^{0.623})$.

However, if I relax my assumption that the cut segments are equal, the recurrence changes. In the most extreme case, the segments run in parallel are only one eighth of the total area. The recurrence for span is then,

$$S(N) = S\left(\frac{N}{8}\right) + 2S\left(\frac{3N}{8}\right) + O(1)$$

To solve this, I will use the substitution method to substitute $S(N)$ with $kN^{\log_a}$ (again, base 2.)

$$S(N) = kN^{\log_a}$$

$$kN^{\log_a} \geq k \left(\frac{N}{8}\right)^{\log_a} + 2k \left(\frac{3N}{8}\right)^{\log_a}$$

$$kN^{\log_a} \geq \left(2 \left(\frac{3}{8}\right)^{\log_a} + \left(\frac{1}{8}\right)^{\log_a}\right) kN^{\log_a}$$

That final equality is satisfied when the term in parenthesis is less than or equal to 1, for which the minimum $a$ is approximately 1.845. Substituting back into the recursion, span is $O(N^{\log_21.845})$ when
\(N\) is area, and since work is \(O(N)\), we have a parallelism of approximately \(O(N^{0.109})\). When \(x, y, t\) are on the order of \(n\), the parallelism is about \(O(n^{0.328})\). However, because the coarsened loops execute in parallel, we get much more parallelism than that in practice.

cilkview returns a parallelism of 15,850 for an input of \(1500 \times 1000 \times 1500\) which is much larger than predicted by either theoretical analysis. I suspect this is because the coarsened loop runs in parallel. I therefore ran cilkview without parallelizing this loop and found a parallelism of 4.55 which is on the order of what the theoretical analysis predicts \(((1000 \cdot 1500 \cdot 1500)^{0.109} \approx 10)\) – some of the lowering being caused by the coarsening which is happening in serial, while my analysis assumed no coarsening.

2.3 Effects of \(T\) Parameter

For the recursive approach, increases in the \(T\) parameter can potentially harm parallelism. If the algorithm is forced to only split along \(t\), there is no parallelism. Because the algorithm must reach a certain ratio of width to height before it can begin space splitting the parallelism of the algorithm is better represented by \(O\left(n^{0.328}\right)/O\left(\log \frac{T}{xy}\right)\). This still does not account for time splits which may occur after the space-time split. To see the effects of these initial time splits, I ran cilkview on both loops and recursive while varying \(t\).

<table>
<thead>
<tr>
<th>X,Y</th>
<th>T</th>
<th>Loops</th>
<th>Recursive</th>
</tr>
</thead>
<tbody>
<tr>
<td>1K</td>
<td>3K</td>
<td>98,555</td>
<td>15,564</td>
</tr>
<tr>
<td>1K</td>
<td>2K</td>
<td>98,555</td>
<td>16,050</td>
</tr>
<tr>
<td>1K</td>
<td>1K</td>
<td>98,555</td>
<td>16,664</td>
</tr>
</tbody>
</table>

Figure 3: Cilkview parallelism results for loops and recursive for varying \(T\)

As the results show, loops has a constant parallelism (since no parallelization occurs in the t-step.) However, the cache-oblivious algorithm’s performance deteriorates slightly with changes in \(T\).

3 Runtime Results with 3-pt Stencils

To compare the runtime performance of loops and recursive, I ran three sets of tests varying \(X, Y\) while holding \(T\) constant. Figures 4 - 6 present the results of testing with \(T = \{1500, 1000, 500\}\)
Figure 4: $T = 1.5K$ The smallest ratio of loops to recursive here occurs at the input 3500 with value 0.73

Figure 5: $T = 1K$ The smallest ratio of loops to recursive here occurs at the input 3500 with value 0.77
The smallest ratio of loops to recursive here occurs at the input 3000 with value 0.79. For large inputs, recursive is about 20% faster than loops. On smaller inputs they perform very similarly. Part of this can be explained by time space splitting, which I will explore more when I increase the stencil size.

4 Larger Stencil Performance

Implementing larger stencils with the recursive algorithm is straightforward. Only sigma, the run of a cut, needs to be increased. This has some huge consequences for performance, however.

4.1 T-Split Consequences of Larger Stencils

The most obvious impact of increasing sigma is the chances of more t-splits, which means fewer parallelized recursions.

4.1.1 Theoretical Consequences

Theoretically, there are few consequences for using a larger sigma. Because we assume in analysis that the recursive splitting maintains the ratio of width to height, once splitting begins, the recurrence should proceed just as in the 3-pt case. However, since the larger sigma requires a larger ratio of width to height, there is a slight increase in the initial splitting. This effect is minimal, though, because it will be in the logarithm, so we will have a theoretical parallelism of \(O(n^{0.328})/O\left(\log \frac{L}{\sqrt{\sigma}}\right)\), which is not much worse than before. So I would expect a constant difference in the number of t-splits.
4.1.2 Actual T-Split Consequences of Larger Stencils

In practice, however, larger stencils perform much worse with the recursive splitting algorithm than the 3-pt stencil. I suspect this happens because the assumption that recursive splitting maintains the ratio of width to height fails for larger sigma. To test this I used an add reducer to count the number of t-splits.

<table>
<thead>
<tr>
<th>X,Y</th>
<th>T</th>
<th>T-Splits for σ = 1</th>
<th>T-Splits for σ = 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>2000</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>500</td>
<td>2000</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>500</td>
<td>1000</td>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>500</td>
<td>500</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 7: T-Split counts for two values of the run, sigma.

The results of Figure 7 show that the number of t-splits for a sigma of 2 is something like $2 \cdot k + 1$ if there are $k$ splits for sigma of 1. This is much worse than the theoretical analysis expected, and may explain why the recursive algorithm performs worse for larger stencils.

4.2 Runtime Results for Large Stencils

The performance gap of the two algorithms closes for larger stencils. I suspect this is because of the t-split problems I discussed in the last section, but there could be other reasons as well.

<table>
<thead>
<tr>
<th>X,Y</th>
<th>Loops (ms)</th>
<th>Recursive (ms)</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>500</td>
<td>1.215</td>
<td>1.393</td>
<td>1.14</td>
</tr>
<tr>
<td>1000</td>
<td>4.165</td>
<td>4.243</td>
<td>1.02</td>
</tr>
<tr>
<td>1500</td>
<td>10.102</td>
<td>10.128</td>
<td>1.00</td>
</tr>
<tr>
<td>2000</td>
<td>19.450</td>
<td>19.598</td>
<td>1.01</td>
</tr>
<tr>
<td>2500</td>
<td>26.812</td>
<td>26.222</td>
<td>0.98</td>
</tr>
<tr>
<td>3000</td>
<td>35.214</td>
<td>33.614</td>
<td>0.95</td>
</tr>
</tbody>
</table>

Figure 8: Runtime results for loops and recursive for a 5-pt stencil.

From Figure 8, I conclude that T-space splitting must play a large part in the slow-down, because as X,Y increases while T is fixed, the performance gap between recursive and loops starts to return to what was observed in the 3-pt case. To test this, I compared their performance for an input of $X,Y = 3000$ and $T = 500$, which results in a ratio of 0.89.
In this lab I investigated the parallelization of a stencil computation to compute the heat equation in a bounded spacetime volume. To parallelize this stencil computation I parallelized a modified version of the cache-oblivious algorithm for stencil computations by Frigo et al. I then examined the performance effects of preemptively partitioning the space further at some iterations, and discovered that this both increased the parallelism of the stencil algorithm in practice. I also performed a rough analysis of the parallelism of the original parallel stencil algorithm and the modified version with an alternative partitioning scheme.

1 Parallelization of Stencil Computation

The cache-oblivious stencil algorithm by Frigo et al. repeatedly partitions the spacetime of the stencil either into two equal space partitions or two time partitions of equal lengths of time. Each space partition divides the space into two equal volume sections such that one partition may be evaluated completely before the other. However, the two space halves generated must be evaluated serially for correctness. In order to parallelize this algorithm we must use a different partitioning scheme.

Our scheme is to perform one partition along both space dimensions at each space partitioning step. Each space partitioning step therefore divides space into four partitions that may be evaluated in the order described in Figure 1. Note that the use of \(-1\) slope dividers for this partitioning step guarantees that the two regions marked “2” in Figure 1 are independent of each other and may therefore be done in parallel.

We can get some intuition for the parallelism we expect from this 4-space division scheme. For simplicity we assume that the division scheme divides the spacetime volume into 4 equal volume partitions. The work for the stencil computation is the work to evaluate all points in the spacetime plus some work to perform the necessary divisions. If we consider a recursion tree for this computation, because each division step takes constant time, the work of the computation equals the number of points in the tree times a constant factor, or \(O(V)\) in total for a spacetime of volume \(V\). For the span of the computation we examine the same recursion tree, but we are able to ignore subtrees of branches that may occur in parallel. Since there is one such branch at every space division represented in the tree, there is at most one ignored branch from each internal node. The total span is therefore at least \(S(V) \geq 3S(V/4) + O(1) = \Omega(V^{\log_4 3})\). Consequently, the parallelism of this scheme is at most \(O(V)/\Omega(V^{\log_4 3}) = O(V^{1-\log_4 3})\).

![Diagram](image.png)

**Figure 1**: Illustrated partitioning of 2 space dimensions into 4 pieces with negative slope cuts. The cuts at the top and the bottom of the trapezoid are shown. The labels within each partition describe the order in which partitions may be evaluated. Two partitions with the same number may be evaluated in parallel.

To increase the parallelism, we note that at each space partition the work does not change, but the span is dependent on the maximum volume touched in a Manhattan walk from one corner of the spacetime to the other. This implies that we could increase the parallelism of our computation by increasing the number of divisions performed at each space-partition step. With this idea in mind I investigated the utility of dividing the spacetime into 16 sections at each space-partition step, as illustrated in Figure 2. Again we use \(-1\) slope partitions to guarantee independence between regions marked with the same number in the figure.

We can use a similar analysis to get some intuition for the parallelism we expect from this 16-space division scheme. We assume that the division scheme divides the spacetime volume evenly for simplicity. We again have
the same amount of work at each step, since the total spacetime volume is unchanged and each division still requires constant work. In the recursion tree for this computation the span need only consider 7 of the 16 branches from each space-partition step. Consequently, we lower bound the span by assuming that at each internal node of the recursion tree our span only executes \( \frac{7}{16} \)ths of the outgoing branches. The span is then at most \( S(V) \geq 7S(V/16) + O(1) = \Omega(V^{\log_{16}7}) \), and the parallelism of this 16-space division scheme is at most \( O(V^{1-\log_{16}7}) \), which is substantially better parallelism than the 4-space division scheme observed.

\[
\begin{array}{cccc}
7 & 6 & 5 & 4 \\
6 & 5 & 4 & 3 \\
5 & 4 & 3 & 2 \\
4 & 3 & 2 & 1 \\
\end{array}
\quad
\begin{array}{cccc}
7 & 6 & 5 & 4 \\
6 & 5 & 4 & 3 \\
5 & 4 & 3 & 2 \\
4 & 3 & 2 & 1 \\
\end{array}
\]

top \quad \text{bottom}

**Figure 2:** Illustrated partitioning of 2 space dimensions into 16 pieces with negative slope cuts. The cuts at the top and the bottom of the trapezoid shown. The labels within each partition describe the order in which partitions may be evaluated. Two partitions with the same number may be evaluated in parallel.

## 2 Experimental Results

I investigated two implementations of the parallel cache-oblivious stencil computation. The first implementation, called “4-space,” used space partitions to divide the spacetime into four volumes executable in the order in Figure 1, and it used time partitions to divide the spacetime into two time partitions that must be executed serially.

The second implementation, called “16/4-space,” checked at each step whether a 16-way space partition was feasible, and performed one if so. If this was not feasible, it would attempt to do a 4-way space partition instead. If this was also infeasible, it would partition along the time axis into four pieces. I used a 4-way time partition for the second implementation in order to make 16-way space partitions more likely to occur.

Note that both implementations used the same computation to compute the dividers as Frigo et al. use for 2-dimensional spacetimes for each space dimension. Consequently these implementations did not divide the volume into equal volume pieces, although such a division is possible.

I ran both algorithms. The raw performance numbers for both implementations are given in Figure 3. On one or two processors the 4-space division scheme was slightly more efficient than the 16/4-space division scheme. This may be due to the additional overheads needed in computing the time and space divisions in the 16/4-space algorithm, which created parallelism that was not effectively used due to the lack of processors. On three or more cores, however, the 16/4-space algorithm noticeably outperforms the 4-space scheme. I believe this is due to the additional parallelism of the 16/4-space algorithm, which is effectively utilized by a larger number of processors. Another benefit of the 16/4-space algorithm is its ability to reduce the depth of the recursion tree, reducing the space and time needed to produce the call stack.

The speedup characteristics for the 4-space and 16/4-space implementations are given in Figure 4 and Figure 5 respectively. As predicted, the 16/4-space algorithm has more parallelism than the 4-space algorithm, and has better speedup than the 4-space algorithm on larger numbers of processors.
<table>
<thead>
<tr>
<th>Processors</th>
<th>4-space (s)</th>
<th>16/4-space (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>74.442</td>
<td>76.614</td>
</tr>
<tr>
<td>2</td>
<td>37.897</td>
<td>38.972</td>
</tr>
<tr>
<td>3</td>
<td>27.495</td>
<td>26.847</td>
</tr>
<tr>
<td>4</td>
<td>21.659</td>
<td>20.277</td>
</tr>
<tr>
<td>5</td>
<td>19.223</td>
<td>17.093</td>
</tr>
<tr>
<td>6</td>
<td>17.584</td>
<td>14.540</td>
</tr>
<tr>
<td>7</td>
<td>16.611</td>
<td>13.075</td>
</tr>
<tr>
<td>8</td>
<td>15.875</td>
<td>11.978</td>
</tr>
</tbody>
</table>

Figure 3: Raw performance numbers for parallel cache-oblivious stencil algorithm using 4-space and 16/4-space partitioning. All tests were conducted on a $4096 \times 4096 \times 512$ spacetime.

Figure 4: Speedup characteristic for 4-space algorithm on $4096 \times 4096 \times 512$ spacetime.

3 Conclusion

I examined two different parallelizations of the cache-oblivious stencil algorithm presented by Frigo et al. I observed that the algorithm that performed more partitioning at each space partition had greater parallelism and better performance on more processors. Further work in this area could involve looking at higher space partitioning schemes and using a balanced space partitioning. Furthermore, a different partitioning scheme, using different slopes of dividers, might allow for additional parallelism by relieving dependencies between additional space partitions. This could result in an algorithm with better parallelism without requiring as many space partitions, and would be an interesting topic of further study.
Figure 5: Speedup characteristic for 16/4-space algorithm on $4096 \times 4096 \times 512$ spacetime.
6.884 Stencil Computation (Laboratory 3)

Zhunping "Justin" Zhang

March, 30, 2010

I have parallelized the code using the following changes:

1. The space cut is replaced by a space + time cut, creating four subproblems. Two of the four subproblems are independent and can be processed in parallel. Theoretically, this gives us a span formula: $T_\infty(n) = 3T_\infty(n/2)$, giving $T_\infty(n) = n^{\log_2 3} = n^{1.59}$ and parallelism $O(n^{0.42})$.

2. Any problem with a time span less than 10 is processed as a leaf.

3. The inner of the leaf are unrolled so that they can be optimized by the compiler to use SSE instructions.

4. The loops code (not the divide-and-conquer code) is improved by replacing the for loop with cilk_for loop and unrolling the inner loop.

The summary of the results are here, using $N = 200$:

<table>
<thead>
<tr>
<th></th>
<th>Given Code (on one worker)</th>
<th>Improved (on eight workers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loops</td>
<td>219ms</td>
<td>190ms</td>
</tr>
<tr>
<td>Divide and conquer</td>
<td>352ms</td>
<td>152ms</td>
</tr>
</tbody>
</table>

The source code is put in heat_parallelized.tgz

1 Extended to Any Stencil

I also made a program routine that takes any form of stencil, including stencils of various shapes and stencil that requires values from earlier than the previous time step. The following code creates a stencil which has the shape of a big cross:

```c
#define TIMEDEPTH 2
extern double u[INITXY][INITXY][TIMEDEPTH]; // we store grid here
#define U(t, x, y) u[x][y][(t)&1]

// total heat added at heat sources per iteration
static const float total_heat_per_frame = 1;

// stencil x_max, y_max
#define KERNEL_MINX -2
#define KERNEL_MINY -2
```
```c
#define KERNEL_MAXX 2
#define KERNEL_MAXY 2

static const int dsx = KERNEL_MAXX;
static const int dsy = KERNEL_MAXY;

/* kernel
   t-1
   ...*...
   ...*...
   .**o**.
   ...*...
   ...*..

* : required value for frame t
o : location of value to be computed */

// kernel
static inline void kernel(int t, int x, int y)
{
    if (x + KERNEL_MINX < 0 || x + KERNEL_MAXX >= X || y + KERNEL_MINY < 0 || y + KERNEL_MAXY >= Y)
    {
        U(t+1,x,y) = 0.0;
    }
    else
    {
        /* this equation might not have any physical meaning,
           just to illustrate the stencil computation */
        U(t+1,x,y) = DTDXDX * (U(t,x+1,y) - 2.0 * U(t,x,y) + U(t,x-1,y))
                     + DTDYDY * (U(t,x,y+1) - 2.0 * U(t,x,y) + U(t,x,y-1))
                     + DTDXDX * (U(t,x+2,y) - 2.0 * U(t,x,y) + U(t,x-2,y))
                     + DTDYDY * (U(t,x,y+2) - 2.0 * U(t,x,y) + U(t,x,y-2))
                     + U(t,x,y);
        // add the heat
        U(t+1,x,y) += heat_inc * raster[y][x];
    }
}

In the main heat program, we also need to make the divide and conquer algorithm know how to cut the space:

#include "common.h"

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) {
        int x, y;
        for (x = x0; x < x1; x++)
            for (y = y0; y < y1; y++)
                kernel(t0, x, y);
    } else if (lt > 1) {
        if (2 * (x1 - x0) + (dx1 - dx0) * lt >= 4 * dsx * lt) {
            int xm = (2 * (x0 + x1) + (2 * dsx + dx0 + dx1) * lt) / 4;
            walk2(t0, t1, x0, dx0, xm, -dsx, y0, dy0, y1, dy1);
            walk2(t0, t1, x0, dx0, x1, dx1, y0, dy0, y1, dy1);
        } else if (2 * (y1 - y0) + (dy1 - dy0) * lt >= 4 * dsy * lt) {
            int ym = (2 * (y0 + y1) + (2 * dsy + dy0 + dy1) * lt) / 4;
            walk2(t0, t1, x0, dx0, x1, dx1, y0, dy0, ym, -dsy);
            walk2(t0, t1, x0, dx0, x1, dx1, ym, -dsy, 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 rect_recursive(int t0, int t1, int x0, int x1, int y0, int y1)
{
    walk2(t0, t1, x0, 0, x1, 0, y0, 0, y1, 0);
}

The dsx and dsy variables tell the recursive program how to cut the space.
The source code is put in anystencil.tgz, the line in common.h determines which stencil to use:

#include "kernel_fivepoints.h"
#include "kernel_ninepoints.h"
#include "kernel_9pts_bigcross.h"
#include "kernel_weird.h"
#include "kernel_5pts_4timedepth.h"