OpenMP Heat Equation

Dr. Christopher Marcotte — *Durham University*

Heat equation

In this exercise we are going to look at an archetypal partial differential equation (PDE) system, the heat equation. This equation models the flow of thermal energy over space and time. First we will consider the steady-states of a one-dimensional version with Dirichlet boundary conditions — this is the simplest version.

A 1D version

In the simplest setting, we have a line segment, with a difference in temperature at either end. We will just set the domain coordinate x to range between 0 and 1. We likewise need an initial condition. The heat equation is then,

$$\partial_t u - \partial_x^2 u = 0, \qquad u(t, x=0) = 0, \quad u(t, x=L) = 1, \qquad u(t=0, x) = u_0(x),$$

where the boundary conditions give us a constant temperature difference, and the initial condition $u_0(x)$ is how the heat is originally distributed. We are interested in the steady state solution, that is, $u(t, x)$ as $t \to \infty$. Note that, in 1D, $\nabla^2 u = \partial_x^2 u$.

Discretise

We will discretise the state u as a one-dimensional array of length N, `double u[N]`, where we identify `u[0]=0.0` and `u[N-1] = 1.0` to satisfy the boundary conditions above. In the interior, we use a three-point stencil to approximate the derivative,

$$\left(\partial_x^2 u\right)_i \approx \frac{u(x_{i+1}) + u(x_{i-1}) - 2u(x_i)}{(\delta x)^2},$$

where $x_{i+1} - x_i = \delta x$ is a constant. This is an example of a finite-difference approximation, and forms the foundation of many numerical approaches for solving PDEs.

Initialize

First we must put the initial values in the array, and we ideally want the values to be

1. smooth to the same order as the underlying equation (should have two spatial derivatives); and
2. respect the boundary conditions, as specified.

For simplicity, I have chosen a hyperbolic tangent which smoothly interpolates between the two boundary values:

heat1D.c

```c
void init(double unew[N]){
  double x;
  unew[0] = cold;
  for (int i = 1; i < N-1; i++){
    x = (float)(i)/(float)(N);
    unew[i] = cold + (hot-cold)*0.5*(1.0 + tanh(10.0*(x - 0.5)));
  }
  unew[N-1] = hot;
}
```

or $u_0(x) = \left(u_0(0) + \left(u_0(1) - u_0(0)\right)\tanh(k(x - 1/2))\right)/2$, with possibly some minor jumps near the actual boundary elements.

Exercise

Try parallelizing the `init(...)` function in ***heat1D.c***. What opportunities for doing so exist? Does parallelizing `init(...)` meaningfully impact the overall runtime of the ***heat1D.c*** program? We could specify the initial condition with a more computationally demanding function (e.g., the Weierstrass function) – would doing so provide more opportunity for parallel speedup of this function?

Solution

There is some opportunity for speeding up `init(...)` – one can create a parallel region, with `private x` per-thread, and evaluate the function for each `i=1` to `i=N-2` with a standard `#pragma omp parallel for`.

The difficulty is what should be done with the first and last elements – we can include them within a larger parallel region by including them in `#pragma omp single nowait` blocks before (or after – it's mostly just important that they appear together) the `#pragma omp for`. Note that the `nowait` clause indicates that the rest of the threads are free to move on, which we can see will work in this case as the mapping from $i \to u_i$ is unique. Alternatively, you can just have a conditional inside the loop argument where when `i == 0` or `i == N-1`, the boundary values are set.

openmp-heat1D.c

```c
double exactSolution(double x, double hot, double cold){
  return cold + (hot-cold)*x;
}

void init(double unew[N]){
  double x;
  #pragma omp parallel default(none) private(x) shared(unew)
  {
    #pragma omp single nowait
    {
    unew[0] = cold;
    }
    #pragma omp single nowait
    {
      unew[N-1] = hot;
    }
    #pragma omp for
    for (int i = 1; i < N-1; i++){
      x = (float)(i)/(float)(N);
```

Now, implementing this has very little impact on the overall performance; if the number of iterations needed to reach tolerance ε is $N(\varepsilon) \gg 1$, then the initialization (which does just a bit more work than a single iteration) makes a contribution which is small, by comparison. If we use a more expensive initialization function, then we can perhaps impact the overall runtime more significantly; in order for the initialization to contribute the same time as the iterations, we would need a function which does about the same amount of work per-element as $N(\varepsilon)$, and then we could improve that runtime by splitting it across p threads. For $\varepsilon = 10^{-6}$, this would mean about 4×10^4 terms in the expansion for the Weierstass function (because computing the power of a floating-point number is more expensive than adding or multiplying). Alternatively, you can think about setting an initial condition which is closer to the true solution so that the number of iterations needed to reach tolerance ε decreases. Generally, we are interested in transformations which increase the amount of work we can do per unit of data. Verdict? Not worth it, but not bad.

Update

Since we are interested in the steady state, we are going to assign $\partial_t u = 0$, and effectively invert the Laplacian, since now $0 = \partial_x^2 u$.

Hint

This is sometimes called a *relaxation* or *annealing* method, and is only valid when seeking a steady state. You may also recognize this as a Jacobi iteration.

Plugging in the original stencil, we see that $u_i = \frac{1}{2}(u_{i+1} + u_{i-1})$. This is replacing the value of the field with a local (weighted) average based on the stencil pattern. In code, this looks like solving `lap_u[i] = 0` for `u[i]`, and assigning that to `unew[i] = (u[i+1] + u[i-1])/2.0;`.

Consider the code for the update below, which updates the values of `unew` according to `u`.

heat1D.c

```c
void step(double unew[N], double u[N]){
  unew[0] = cold;   //unew[0]   = (u[0] + u[1])/2.0;      // alternative boundary conditions
  for (int i=1; i < N-1; i++){
    unew[i] = (u[i-1] + u[i+1])/2.0;
  }
  unew[N-1] = hot;  //unew[N-1] = (u[N-1] + u[N-2])/2.0;  // alternative boundary conditions
}
```

Exercise

Parallelise the update function `step(...)` using a straight-forward `#pragma omp parallel for` and time the execution with 1, 2, 4, 8 threads using OpenMP to get a baseline strong scaling. Estimate the parallel fraction. Is this parallelism worthwhile? What might be changed in this program to generate sufficient work for the parallelisation of `step(...)` to be worthwhile (up to 8 threads)?

Solution

You will almost certainly find that the parallelism is detrimental in this case, at least for values of N which we will consider here.

For $N = 2^{10}$, I found times which increased with p; while for $N = 2^{20}$, I found times which decreased with p. The summary times for $p = 1, 2, 4, 8$ on my laptop are in Table 1 and Table 2 for $N = 2^{10}$ and $N = 2^{20}$, respectively. For the large domain, the parallel fraction is roughly $f \approx 0.14$.

p	t	**Iterations**
1	0.8613889999687672	246611
2	3.6712969999643974	246611
4	6.6260619999957271	246611
8	11.7048660000436939	246611

Table 1: Timing results for $N = 2^{10}$ and $p = 1, 2, 4, 8$. This works out to just over 3 ns per element per iteration for $p = 1$.

p	t	**Iterations**
1	1.4542500000097789	440
2	0.8281699999934062	440
4	0.5029069999582134	440
8	0.3613599999807775	440

Table 2: Timing results for $N = 2^{20}$ and $p = 1, 2, 4, 8$. This works out to just under 1 ns per element per iteration for $p = 8$.

Copy

In the serial code, we copy the values from `double* unew` into `double* u`. This is done manually, and follows the same pattern as all the other functions – a loop which maps an index uniquely to a position in memory.

Exercise

Parallelize the function `copy(...)`, ensuring the consistency of the data throughout. For this function, use the `num_threads(int)` clause to specify the number of threads and determine for which value of threads the copy is fastest. Use the `omp_get_wtime()` function to determine the copy timing. How might your results change for large N?

Solution

You will likely find that this is both trivially parallelized and not worthwhile – turns out the loop contains so little work that it's better to not parallelize it *at all*. Indeed, on my machine I found the overhead of even using the `#pragma omp ...` was sufficient to make the copy substantially slower. If you are able to speed up the copy, it is likely due to a low number of threads coupled with a very large N, and even then you are likely just barely faster. An advanced option which we have not really dealt with in this module is to use SIMD, i.e. `#pragma omp parallel for simd` which could improve things if your compiler hasn't automatically vectorized the copy.

Convergence

To test when the solution has converged (stopped changing), we will measure the maximum difference in the state array `double* u` compared to `double* unew`. To this end, we store this maximum in a single variable and compare it to a small threshold for the convergence of the solution:

heat1D.c

```c
double diff(double unew[N], double u[N]){
  double maxdiff = 0.0;
  for (int i=0; i < N-1; i++ ){
    if (maxdiff < fabs(unew[i]-u[i])){
      maxdiff = fabs(unew[i]-u[i]);
    }
  }
  return maxdiff;
}
```

Exercise

Implement the parallel calculation of `double maxdiff` using a `#pragma omp reduction` construct. Refer to the example ***testmax.c*** if you're confused on the calling signature (function call v. operator).

Solution

This is a straight-forward application of the reduction construct, but perhaps obscured by the use of the maximum operator (instead of our usual addition).

openmp-heat1D.c

```c
        unew[i] = (u[i-1] + u[i+1])/2.0;
      }
    }
    #pragma omp for
    for (int i=1; i < N; i+=2){
      if (i == 0){
        unew[i] = cold;
      }else if (i == N-1) {
```

Coloring

You perhaps found the parallelization of the 1D code pretty easy and also found it yields little speedup. Parallelising this update is somewhat less trivial than our previous efforts, since we have cross-index dependencies; index `i` in `unew` depends on `i+1` and `i-1` in array `u`. One of the ways this has been handled is by *coloring*, where each index is assigned a color so that all updates which do not interact can proceed simultaneously. Then the procedure repeats until the entire update is complete. This stencil has a convenient coloring — if `i` is even, then `i+1` and `i-1` are odd, so we can specify a coloring using the parity of index `i`. This is usually called a "red-black" coloring.

Exercise

Try to implement the coloring by splitting the `step(...)` function loop into two: one for i even and the other for i odd. Ensure you are only reusing the threads by separating the `parallel` and `for` clauses. Time the execution with an increasing number of OpenMP threads and compare to the baseline strong scaling you found before. Estimate the parallel fraction. Is it significantly changed?

Solution

In principle, red-black Jacobi iterations can be a substantial speedup if we are compute-bound and the cost of traversing the array twice is an insignificant proportion of the runtime. In practice, I was unable to find a configuration which actually sped things up compared to the baseline, and estimate $f \approx 0.17$.

p	t	**Iterations**
1	1.5077390000224113	440
2	0.9327799999155104	440
4	0.6240209999959916	440
8	0.4256989998975769	440

Table 3: Timing results for $N = 2^{20}$ and $p = 1, 2, 4, 8$ with red-black updates. This works out to just about 1 ns per element per iteration for $p = 8$.

Boundary Conditions

The boundary conditions we've used here are convenient, but not very interesting. In many applications, it is better justified to say that there is no *flux* of heat through a boundary. In one dimension, we write $\partial_x u(x = 0) = 0$, and in more than one we write $\hat{n} \cdot \nabla u = 0$ — read as 'the part of the gradient of u which is normal to the boundary vanishes'. In our implementation, this means that `u[0]` and `u[N-1]` are no longer fixed, and instead vary over time. The standard update rule, `unew[i] = (u[i-1] + u[i+1])/2.0;`, will not work when `i==0` or `i==N-1`, so we need an update rule for these indices.

The 'no-flux' conditions are sometimes referred to as Neumann – a condition on the derivatives – whereas the boundary conditions we started with are referred to as Dirichlet – a condition on the state itself.

To this end we will use a fictional extra index past the range of the array, `u[-1]` and `u[N]`, where the value exactly mirrors the interior neighbor value: `u[-1] = u[0]` and `u[N] = u[N-1]`. Techniques like these are sometimes called *ghost cells*. Forming the update rule from above and plugging in these identities, we find update rules for the edge values: `unew[0] = (u[0] + u[1])/2.0;` and `unew[N-1] = (u[N-1] + u[N-2])/2.0;`.

Challenge

Implement these new boundary conditions in your 1D code. How does the solution change? How do the number of iterations before reaching convergence change? Why? Can you readily compare the performance of this new model (because of the changed boundary conditions) to the performance of the old model? What limitations are there to the comparison? What new information do you need to describe?

Hint

Try plotting the solutions to better understand why one may be substantially harder than the other.

A 2D Example

The heat equation is, as stated above, archetypal. Everyone has done it, and its a great exercise for optimization with OpenMP. You may have noticed that the one dimensional system does not give your computer a workout and hardly benefits from our parallelisation efforts. The two dimensional system should show somewhat more compelling strong scaling. Look at the ***heated_plate_new.c*** code, which I adapted from a code published by John Burkardt at Florida State University.

Challenge

Compile the code using `gcc` and run the executable, making note of how many iterations it requires to achieve the prescribed tolerance.

Time it using the unix command, `time`, or add calls to time it like you did in 1D.

Compare the adapted code ***heated_plate_new.c*** to John's original code, and to John's OpenMP parallelised implementation.

At the time of John's implementation, OpenMP could not handle `#pragma omp reduction` clauses using a `max` operator in *C* (as John notes in his OpenMP code). Adapt John's OpenMP code to perform a `reduction` using `max` for the `diff` variable, instead of his version with `my_diff` intermediates. Do you get the same results? Time the original code and the new code. Is it faster to use a `reduction` clause?

Adapt the parallelism from John's OpenMP code to ***heated_plate_new.c***. Do you need to restructure ***heated_plate_new.c*** to maximize the parallelism? Compare your OpenMP parallelised code to John's.

We will return to the heat equation when discussing MPI, so it is a good idea to make yourself familiar with some of the issues with solving and parallelising it.

Aims

- Implementation of a system of ODEs (or a discretized PDE) with `C` and OpenMP
- Introduction to some standard techniques for parallelism of these types of calculations
- Convergence testing as an example of inter-thread communication (reduction)
- Practice with reading and extending existing `C` codes
- Critical assessment of similar codes
- Practice with updating codes to newer standards