The Heat Equation:

High-Performance Scientific Computing Case Study

Micah D. Schuster
Wentworth Institute of Technology

Editor:
Nargess Memarsadeghi, nargess.memarsadeghi@nasa.gov

The need for high-performance computing (HPC) continues to grow as new segments of industry and academia look to solve complex problems with large data requirements. Thus, the need for skilled programmers trained in specific problem domains and with knowledge of using supercomputers is at an all-time high.

Typical high-performance applications strive to run on many different computer architectures, from a laptop to the largest supercomputers in the world. This is a daunting task that requires detailed information about the problem at hand, careful design, and access to performance analysis tools. Many supercomputing centers offer an initial allocation, typically around 100,000 CPU hours, to allow potential users to evaluate the performance of their application. Larger allocations are then given via a competitive proposal process.

Learning the basics of parallel programming is usually done in a classroom setting using the small- to medium-sized supercomputers found in university departments. This is very useful for the distributed memory programming, typically using an implementation of the message passing interface (MPI) standard, where computationally intensive parts of an application are spread among multiple nodes in a cluster. While my future case studies might tackle this topic, our focus in this column will be on shared memory parallelism. This is easily performed on any multi-core desktop or laptop by utilizing two or more cores to their full potential.

The heat equation that we’ll model is important in many scientific fields. In mathematics, the prototypical parabolic partial differential equation can be used to describe a wide variety of time-dependent phenomena, including heat conduction, particle diffusion, and ocean acoustic propagation. A direct practical application of the heat equation, in conjunction with Fourier theory, is the prediction of thermal transfer profiles and the measurement of the thermal diffusivity in polymers. In financial mathematics, the famous Black–Scholes option pricing model’s differential equation can be transformed into the heat equation, allowing relatively easy solutions from a familiar body of mathematics. The heat equation can also be used to perform edge detection that can easily be parallelized due to local operations performed over the image. The diffusion equation, a more general version of the heat equation, arises in connection with the study of chemical diffusion and other related processes.

This column is divided into two parts: an introduction to the terminology and tools that can be used to parallelize an application, and a description of the process to derive a numerical scheme, program, and parallelize the heat equation. I start with HPC terminology and the basics of shared memory programming using an API called OpenMP. For the solving heat equation, I first derive the numerical scheme that we’ll program, and then describe how to parallelize the algorithm using OpenMP.
HPC CONCEPTS

As with any field, a variety of terms and concepts are worth exploring before we continue on to the parallelization and analysis of the heat equation. Chief among these concepts is the parallel programming model that we’ll be using. Since most users do not have access to larger supercomputer systems, the shared memory model will be ideal. This model assumes that multiple cores have access to the same large block of memory. Most modern computers that have multicore processors will fit this scenario. Typically a given application only uses one core at any given time, thus executing the program instructions sequentially, that is, in serial. Our goal will be to utilize multiple cores, which share the same system memory, to solve a given problem in a shorter amount of time.

We typically think of a processor core as a hardware unit that executes a stream of instructions in serial. This stream is associated with a single application and is typically called a thread. Multicore processors are useful because we always have many applications running on our system, from applications managed by our OS to multiple tabs in a web browser. Shared memory programming has two distinct features: utilizing more than one thread in a single application, which allows instructions to be executed simultaneously; and each thread having access to the entire system memory. Threads interacting with the same memory location at the same time can cause complications, but in general, shared memory programming is the quickest way to utilize a multicore system.

When we create a parallel program, our goal is to use more than one of the available cores to reach the solution to our problem. Ideally, we should arrive at the solution to the same problem faster than when using a single core. A key concept in determining your parallel code performance is speedup, given by

\[
\text{Speedup} = \frac{\text{Serial Runtime}}{\text{Parallel Runtime}}.
\]

The runtime of the code is usually computed using wall clock time, similar to using a stopwatch to time different parts of the code. In an application, you use API calls that report the current time relative to some point in the past, generally related to when the system was powered on. We can then examine the runtime of our entire code or limit the investigation to computationally intensive sections.

Parallel efficiency uses the speedup to give a measure of how well the code has been parallelized. Efficiency is given by

\[
\text{Efficiency} = \frac{\text{Speedup}}{p},
\]

where \( p \) is the number of threads used to solve the problem, not exceeding the number of cores on the system. As an example, if a specific problem sees a speedup of 1.7 when utilizing 2 cores/thread, the efficiency is 85 percent. In a typical application, the efficiency tends to decrease as the number of threads increase, which is due to bottlenecks in the computational algorithms themselves or other areas of the code. Achieving 100 percent efficiency is not usually possible, but certain types of applications can come close.

The last major concept, scalability, is related to speedup and efficiency, but focuses more on the size of the problem that one is attempting to solve and the time it takes to do so. In parallel computing, scalability refers to how the runtime of the application changes as the number of processors increase. There are two main types:

- **Strong scaling.** Performing a strong scaling analysis uses the performance of a parallel code with a fixed problem size as the number of processors vary. Ultimately, this tells us information about parallel overhead; the time it takes to coordinate parallel tasks, for example, create and destroy threads; and the parallel algorithm efficiency as more processors are used to solve the problem.
• **Weak scaling.** A weak scaling analysis uses a fixed amount of work per processor and compares the execution time as the number of processors vary. This is a subtle but important contrast to strong scaling: as you increase the number of processors for a weak scaling test, the problem size must also get proportionally bigger. As a result, each processor will always have the same amount of work to perform even as the number of processors increases. Weak scaling is often used for determining how large a problem can get while still maintaining the same parallel efficiency.

Regardless of the type of scaling you test, there are several important considerations regarding the testing process. Scaling tests should be done on realistic problem sizes and for realistic lengths of time. Generally, there will be some overhead at program initialization and during the computationally intensive sections of code. If the problem size is too small, the overhead might be a significant fraction of the work, and the program will appear to have very poor scaling.

**OPENMP INTRODUCTION**

OpenMP is a specification for a set of compiler directives, library routines, and environment variables that can be used to implement parallelism in Fortran and C/C++ programs. To parallelize code, programmers look for regions where instructions can be shared among processors. Most often, the focus is on distributing loop iterations among threads, that is, giving each thread a subset of the total loop iterations and then combining the result in the desired way when finished. This is known as loop-level parallelism and will be the method that we use to parallelize the heat equation. Typically, the thread that starts the program execution forks a specified number of worker threads, known as a thread team, and the system divides a task among them. This main thread, also known as the master thread, is assigned an ID of 0 in the team. The threads then run concurrently, with the runtime environment allocating threads to different processors. This is the fork-join model, illustrated in Figure 1. Typically, multiple regions of code will be parallelized in this fashion.

![Figure 1. The fork-join model is characterized by an initial master thread forking into multiple worker threads at specified areas called parallel regions. After the parallel regions finish, the threads rejoin into a single master thread and serial execution continues. The parallel regions are not restricted to a set number of threads or a set length.](image)

Most C/C++ and Fortran compilers have a built-in OpenMP implementation. The free compiler suite that we’ll be using is the GNU Compiler Collection (GCC). To use the OpenMP runtime library, simply include the header file `#include <omp.h>`. The compiler directives can be used by including the `-fopenmp` compiler option when compiling the source code.

Since OpenMP is largely compiler-directive-based, its implementation in preexisting code does not require large sections to be rewritten. As a result, we’ll take our serial algorithm and add directives in the appropriate locations to achieve shared memory parallelism. I’ll start by defining the important OpenMP directives, in C, that we need to know to successfully parallelize the heat equation.

As an introduction to using OpenMP, I’ll first describe the important directives that we’ll use for parallelizing the heat equation. Then, we’ll look at a simple linear algebra operation, the dot product, to illustrate the process of going from a serial code to a parallel code by using OpenMP...
Important Directives

The most important directive in OpenMP creates multiple threads that execute the code within a specified region:

```
#pragma omp parallel num_threads(2)
{
    //code block to be executed by multiple threads
}
```

#pragma omp is known as the sentinel, and lets the compiler know that the following line contains code that should be interpreted as an OpenMP directive. `parallel` is the directive that creates the parallel region between the following braces. Lastly, `num_threads(2)` is a clause that instructs the compiler that the parallel region should create a specific number of threads, two in this case. If the `num_threads` clause is not included, the default value is the number of cores on the system. If the integer value is invalid, for example, zero or negative, the program will fail to compile. However, entering a value that exceeds the number of cores on the system will work correctly, although the number of threads that can run simultaneously is limited by the number of physical cores available.

By itself, creating a parallel region won’t result in faster code—it just means that multiple threads will be performing the exact same tasks. This is where one of the next most common directives comes in:

```
#pragma omp parallel num_threads(2)
{
    #pragma omp for
    for( int i = 0; i < N; i++ ){
        //work to do
    }
}
```

The `for` directive is the easiest way to achieve loop level parallelism when using OpenMP. When the threads in the parallel region encounter the `for` directive, the loop iterations are distributed to the threads. In the example above, two threads will split the iterations numbered 0 through $N$ as evenly as possible, causing both threads to get $N/2$ iterations to compute. Note that you can’t assume that a specific thread will get a specific group of iterations, as this behavior is determined by the compiler implementation of OpenMP, not by the standard.

The variables within the parallel regions, $i$ and $N$, have important properties assigned to them. $N$ is a shared variable because it is, presumably, declared before the creation of the parallel region. This means that each thread will access the same value in memory, effectively sharing the value. The loop variable, $i$, on the other hand, is defined within the parallel region. This makes $i$ private, meaning that each thread will have its own copy of the iterator. By contrast, if $i$ were shared, multiple threads would be using and possibly changing the value of $i$ at the same time, causing incorrect results during execution.

There are clauses that can be used along with the `for` directive to specify the status of incoming variables. For example, if we needed $N$ to be private, even though it was declared before the parallel region, we could use the clause `private(N)` after the `parallel` directive or after the `for` directive. Within the private clause, you can use a list of as many variables as needed. Similarly, there is a `shared` clause that works the same way.
Dot Product

As an example of using the above directives for a realistic calculation, we can tackle one of the most straightforward linear algebra operations, the dot product. The dot product is an operation that takes two vectors and returns a scalar,

\[ \mathbf{a} \cdot \mathbf{b} = \alpha, \]

where \( \mathbf{a} \) and \( \mathbf{b} \) can be any size vector. Computing the dot product can be done using a summation of the products of each element in vectors of the same size, \( N \),

\[ \mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{N} a_i b_i. \]

The full code, listed below, is simply a loop over the size of the arrays:

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main(int argc, char **argv){
    int sum = 0;
    int N=100;
    int a[N], b[N];

    //initialization
    for( int i = 0; i < N; i++ ){
        a[i] = b[i] = 1;
    }

    //dot product
    for( int i = 0; i < N; i++ ){
        sum += a[i] * b[i];
    }

    printf("Dot Product: %d\n",sum);
    return 0;
}
```

This can be compiled using the GCC C compiler:

```
gcc -fopenmp <FILENAME> -o <EXE>
```

There are two OpenMP specific parts, one in the source code and one for the compilation. `#include <omp.h>` allows for use of the OpenMP runtime library and the `-fopenmp` compiler option is used to activate any compiler directives included in the source code.

After iterating through the loop, the scalar result of the dot product will be contained in the variable `sum`. This calculation can be parallelized with the directives shown above and careful thought about what variables should be shared and private.

```c
#pragma omp parallel private(sum)
{
    #pragma omp for
    for( int i = 0; i < N; i++ ){
        sum += a[i] * b[i];
    }
}
```

On first glance, this code appears to be parallelized correctly; however, since the variable `sum` is set to private, each thread will have its own copy. This is certainly preferable to letting `sum` be
shared, as all the threads would be attempting to change the variable at any given time. As written, each thread will contain a small portion of the total result, but there is nothing written that combines all the thread’s values contained in `sum` into a single number. Fortunately, we don’t have to create this functionality from scratch. OpenMP provides a clause specific to the `for` directive that will do this process for us.

```c
#pragma omp parallel
{
    #pragma omp for reduction(+:sum)
    for( int i = 0; i < N; i++ ){
        sum += a[i] * b[i];
    }
}
```

The `reduction` clause is used to combine all thread values of a specific private variable into one value at the end of the `for` loop. In our case, we want to perform the addition operation on the `sum` variable, `(+:sum)`. The general syntax is `reduction(op:variable list)`. There are several different types of allowed operations and any number of variables are allowed in the list. However, pointers can’t be used, which limits the reduction to single values only. Note that a `private` clause is no longer associated with `sum`, because any variable that exists within a `reduction` clause is automatically private.

The last aspect of parallelizing any code is analyzing the performance of the serial version and the parallel version. To do this, we need some way to time the important regions of the code. The OpenMP runtime library includes the function `double omp_get_wtime()`, which will return a value, in seconds, from some arbitrary point in the past. By calling this function before and after the region of interest and subtracting the values, you can obtain the wall clock time.

```c
double start = omp_get_wtime();
#pragma omp parallel
{
    #pragma omp for reduction(+:sum)
    for( int i = 0; i < N; i++ ){
        sum += a[i] * b[i];
    }
}
double end = omp_get_wtime();
double elapsed = end - start;
```

It is worth timing any regions that might be time intensive, and examining their performance while changing the number of threads. This timing data can then be used to create a figure showing a strong scaling analysis of the algorithm. Usually, timing the entire parallel region is preferable; however, it can be useful to put these calls within parallel regions to gather timing data for individual threads.

**Activity 1: Dot Product**

1. Code the parallel dot product using the code listings above.
2. Instead of statically allocating the arrays `a` and `b`, use the `malloc` function (or `new` keyword) to dynamically allocate memory. Be sure to initialize the arrays properly and deallocate the array memory when finished.
3. Use `omp_get_wtime()` to compute the elapsed wall clock time for the main dot product loop.
4. Examine how this execution time changes as you increase the number of threads. Use values up to and beyond the number of processor cores on your system.
For small values of $N$, there is not enough work to perform to justify the overhead cost related to creating and managing the threads. However, for large values of $N$, this code will perform significantly better than the serial version. As an example of a linear algebra operation that is more computationally intensive, we can look at a matrix-vector multiplication, which can be represented as a series of dot products. In this case, because of the additional work to be done, the overall parallel performance will be higher.

**Activity 2: Matrix-Vector Product**

1. The matrix-vector product for an $m \times n$ matrix can be computed as

   \[
   \begin{bmatrix}
   a_{00} & a_{01} & \cdots & a_{0n} \\
   a_{10} & a_{11} & \cdots & a_{1n} \\
   \vdots & \vdots & \ddots & \vdots \\
   a_{m0} & a_{m1} & \cdots & a_{mn}
   \end{bmatrix}
   \begin{bmatrix}
   h_0 \\
   h_1 \\
   \vdots \\
   h_n
   \end{bmatrix}
   =
   \begin{bmatrix}
   a_{00}h_0 + a_{01}h_1 + \cdots + a_{0n}h_n \\
   a_{10}h_0 + a_{11}h_1 + \cdots + a_{1n}h_n \\
   \vdots \\
   a_{m0}h_0 + a_{m1}h_1 + \cdots + a_{mn}h_n
   \end{bmatrix}.
   \]

2. This can be interpreted as a series of dot products. Modify the dot product code from Activity 1 to compute a matrix-vector product.

3. Parallelize this matrix-vector product and perform an analysis on the execution time versus number of threads for several different matrix dimensions.

**THE HEAT EQUATION**

We’ll start our investigation with the 1D equation,

\[
\frac{\partial \phi}{\partial t} = \alpha \frac{\partial^2 \phi}{\partial x^2},
\]

where $0 \leq x \leq L$ and $t \geq 0$. This models the heat conduction through a material with thickness $L$ and a thermal diffusivity represented by $\alpha$, which is typically a constant coefficient. $\alpha$ measures the ability of a given material to conduct heat; the better a material can conduct heat, the larger the value will be. The units chosen for $\alpha$ determine the units for the rest of the values in the problem. Here we’ll use the SI units, m$^2$/s. The units for the temperature, $\phi$, are typically in kelvin (K).

The solution requires one to specify the boundary conditions at $x = 0$ and $x = L$ and the initial state of the material at $t = 0$. For our proposes, a fixed value at the spatial boundaries, known as Dirichlet conditions, and an initial function at $t = 0$ will suffice. More complicated boundary conditions can be used; however, that is beyond the scope of this case study.

**Finite Difference Approximation**

The finite difference method is a technique for obtaining the solution to ordinary and partial differential equations (PDEs). Rather than a continuous solution, these equations can be solved on a discrete grid that represents the domain of interest, such as a slab of some material. The number of grid points is generally chosen by the user and depends on the problem at hand and the computational power available. In general, increasing the number of points increases the resolution of the grid and the accuracy of the numerical solution.

Discretizing the domain results in a set of algebraic equations for the discrete unknowns that can easily be solved on a computer. The main idea is to replace the continuous derivatives with finite difference formulas that only use the discrete values associated with the grid points.

For the heat equation, there are two partial derivatives. Each is replaced with a discrete difference formula that uses different combinations of grid points, this results in the finite difference scheme for solving our PDE. As a consequence of how we derive the numerical scheme, if the grid spacing in both space and time goes to zero, the numerical solution will approach the continuous solution. We’ll focus on the Forward-Time, Centered Space (FTCS) scheme to solve the
heat equation. There are many schemes that one could use for solving the heat equation that can be found online and in many textbooks, where more rigorous mathematical treatments can be found. For our purposes, we’ll focus on understanding the implementation and numerical accuracy of the two schemes.

We’ll solve the heat equation on a uniform grid for a finite set of $x$ and $t$. The discretization scheme yields a few simple equations for both time and space. Each spatial grid point will be separated by

$$\Delta x = \frac{L}{N-1},$$

where $L$ is the length of the domain and $N > 1$ is the total number of points. Using this distance, the grid points, $x_i$, can be computed using:

$$x_i = (i-1)\Delta x, \quad i = 1, 2, \ldots N.$$

Similarly, time can be discretized in the same way. If $t_{\text{max}}$ is the maximum time allowed and $M > 1$ is the number of desired time steps, the size of each step is given by

$$\Delta t = \frac{t_{\text{max}}}{M-1}$$

and the temporal grid points are given by

$$t_m = (m-1)\Delta t, \quad m = 1, 2, \ldots.$$

The last step before we can develop a scheme to solve the heat equation is to understand how to numerically compute a derivative. Finite difference methods involve using a discrete approximation such as

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x}.$$  (2)

In this notation, $\phi_i$ represents the numerical value at the grid point $x_i$. One way to conceptualize this approximation is to consider the definition of a derivative:

$$\frac{d f}{d x} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}.$$

This represents a secant line connecting two points, $f(x + h)$ and $f(x)$, as their separation decreases. Geometrically, a derivative is the tangent line at a point on a curve, as $h$ approaches zero, the secant line becomes the tangent line.

In general, we choose $\Delta x$ from Equation 2 to be as small as possible. The accuracy of this approximation, in addition to a mathematical derivation, can be examined via Taylor series expansion. If we expand $\phi(x)$ about a grid point, $x_i$,

$$\phi(x_i + \Delta x) = \phi(x_i) + \Delta x \frac{\partial \phi}{\partial x} + \frac{\Delta x^2}{2} \frac{\partial^2 \phi}{\partial x^2} + \frac{\Delta x^3}{3!} \frac{\partial^3 \phi}{\partial x^3} + \ldots.$$

Solving for $\frac{\partial \phi}{\partial x}$ gives us nearly the same expression as Equation 2 with a few extra terms:

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x}.$$  (2)

By ignoring the higher-order derivatives and substituting the approximate solution for the exact solution, $\phi_i = \phi(x_i)$ and $\phi_{i+1} = \phi(x_i + \Delta x)$, we get

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_i}{\Delta x}.$$
The magnitude of the truncated terms will depend on \( \Delta x, \Delta x^2 \), and so on. We have a great deal of control over the size of the truncated terms and assuming that we choose a small value for \( \Delta x \), these terms will have a small, but nonnegligible, influence on our solution. We can then use big \( O \) notation to represent the truncation error associated with our approximate solution:

\[
\frac{\partial \phi}{\partial x}|_{i} = \phi_{i+1} - \phi_{i} + O(\Delta x). 
\]

This equation is known as the forward difference approximation for \( \frac{\partial \phi}{\partial x} \) because it uses the \( x_i \) and \( x_{i+1} \) grid points.

We can obtain a similar finite difference formula by using \( -\Delta x \) in the Taylor expansion. The result is known as the backward difference approximation:

\[
\frac{\partial \phi}{\partial x}|_{i} = \phi_{i} - \phi_{i+1} + O(\Delta x). \quad (3)
\]

The order of the truncation error is the same as the forward difference approximation.

**Activity 3: Derivation of the Backward Difference Formula**

1. Use the Taylor series expansion to derive the backward difference formula (Equation 3).

Since both difference formulas give us the same truncation error, can we derive a formula for \( \frac{\partial \phi}{\partial x} \) that gives us a smaller error? In the previous derivations, we use the Taylor series expansion about \( i - \Delta x \) or \( i + \Delta x \). However, there is one other way that we can approximate the derivative, a central difference, where we use both and compute their difference. Let’s compute the Taylor series expansions for \( \phi_{i+1} \) and \( \phi_{i-1} \):

\[
\phi_{i+1} = \phi + \Delta x \frac{\partial \phi}{\partial x}|_{i} + \frac{\Delta x^2}{2!} \frac{\partial^2 \phi}{\partial x^2}|_{i} + \frac{\Delta x^3}{3!} \frac{\partial^3 \phi}{\partial x^3}|_{i} + \ldots,
\]

\[
\phi_{i-1} = \phi - \Delta x \frac{\partial \phi}{\partial x}|_{i} + \frac{\Delta x^2}{2!} \frac{\partial^2 \phi}{\partial x^2}|_{i} - \frac{\Delta x^3}{3!} \frac{\partial^3 \phi}{\partial x^3}|_{i} + \ldots.
\]

The central difference, which uses the grid points directly surrounding \( i \), is

\[
\phi_{i+1} - \phi_{i-1} = 2\Delta x \frac{\partial \phi}{\partial x}|_{i} + \frac{2\Delta x^2}{3!} \frac{\partial^3 \phi}{\partial x^3}|_{i} + \ldots, \quad (4)
\]

and solving for the first derivative yields

\[
\frac{\partial \phi}{\partial x}|_{i} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} + \frac{2\Delta x^2}{3!} \frac{\partial^3 \phi}{\partial x^3}|_{i} + \ldots,
\]

\[
\frac{\partial \phi}{\partial x}|_{i} = \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} + O(\Delta x^2).
\]

When choosing a small value for \( \Delta x \), the truncation error goes to zero much faster than the forward or backward difference formulas.

The heat equation contains a first-order derivative in time and a second-order derivative in space. For time, we must use either the forward or backward difference formula because the central difference would require the previous time step and the next time step to compute the current step. The central difference would work well for the spatial derivative, but we first need a numerical approximation for a second derivative. By manipulating the Taylor series, we can derive the second-order approximation. Instead of subtracting in the left side of Equation 4, we can add

\[
\phi_{i+1} + \phi_{i-1} = 2\phi + \Delta x^2 \frac{\partial^2 \phi}{\partial x^2}|_{i} + \frac{2\Delta x^2}{4!} \frac{\partial^4 \phi}{\partial x^4}|_{i} + \ldots.
\]
Solving for the second derivative, we get the second-order central difference approximation:

\[
\frac{\partial^2 \phi}{\partial x^2} \bigg|_{x_i} = \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2} + O(\Delta x^2).
\]

**Forward-Time, Centered Space**

With the formulas representing the first and second derivatives in hand, we can now create a discrete scheme to solve the heat equation. Because we’ll be considering grid points in both time and space, we need to add a superscript \( n \), in addition to the \( i \) subscript for the spatial grid point, to our notation to denote the current time step. The heat equation contains a first-order derivative in time, the spatial points are held constant, which we can approximate by

\[
\frac{\partial \phi}{\partial t} \bigg|_{x_i,t} = \frac{\phi_{i}^{n+1} - \phi_{i}^{n}}{\Delta t} + O(\Delta t).
\]

Notice that the approximation only depends on points located on the current grid point: \( x = x_i \).

For the second-order spatial derivative, we can use the second-order central difference approximation with the added time notation

\[
\frac{\partial^2 \phi}{\partial x^2} \bigg|_{x_i} = \frac{\phi_{i+1}^n - 2\phi_i^n + \phi_{i-1}^n}{\Delta x^2} + O(\Delta x^2).
\]

The grid on which this solution will be solved can be visualized in Figure 2.

![Figure 2. Grid for solving the heat equation. The filled squares represent the known quantities, the initial condition at \( t = 0 \) and the boundary conditions at each time step.](image-url)

Substituting these two approximations into Equation 1, we arrive at

\[
\frac{\phi_{i}^{n+1} - \phi_{i}^{n}}{\Delta t} = \frac{\phi_{i}^{n+1} - 2\phi_i^* + \phi_{i}^{n}}{\Delta x^2} + O(\Delta t) + O(\Delta x^2).
\]

Note that the temporal and spatial errors have different orders.

The truncation terms won’t be included in our final finite difference scheme, but they are important to consider when choosing values for the time step and spatial grid spacing. The overall goal of this approximation is to update each grid point from the current time step to the next. In the above equation, there is one term that represents that: \( \phi_{i}^{n+1} \). Rearranging the equation yields the FTCS approximation for the heat equation:
\[
\phi^{n+1} = \phi^n + \frac{\alpha \Delta t}{\Delta x^2} (\phi^n_{i+1} - 2\phi^n_i + \phi^n_{i-1}).
\] (5)

This scheme is easy to implement because the values of each grid point at the next time step, \(\phi^{n+1}\), can be updated independently of any other grid point in that next time step. This can be represented by a stencil, shown in Figure 3.

Figure 3. Computational stencil for the FTCS method.

Using this scheme requires two additional pieces of information: an initial condition, which represents the state of the grid at \(t = 0\); and a boundary condition at each end of the grid, \(x_0\) and \(x_L\). With these values set, the simulation is performed by iterating through Equation 5. The FTCS method can yield solutions that become unstable if the value of \(\Delta t\) is too large. These instabilities manifest as oscillations that grow over time. Stable solutions can be obtained under the following condition:

\[
\frac{\alpha \Delta t}{\Delta x^2} < \frac{1}{2}.
\]

The implementation of the FTCS scheme to solve the heat equation can be done via a nested loop structure as outlined below.

```c
float r = alpha*dt / ( dx*dx );
for( int t = 0; t < tmax; t++ ){
    for( int i = 0; i <= N; i++ ){
        phiNew[i] = phiOld[i] + r*(phiOld[i+1] - 2*phiOld[i] + phiOld[i-1] );
    }
    //save or output new value
    phiOld = phiNew;
}
```

The arrays \(\phi\text{New}\) and \(\phi\text{Old}\), presumably declared and initialized previously in the program, represent the two time steps that are necessary to perform the calculation, \(\phi^{n+1}\) and \(\phi^n\), respectively, and their array elements represent the \(x\) position on the grid. The values for \(\alpha\), \(dt\), \(dx\), and \(tmax\) represent the thermal diffusivity, time step, spatial step, and max time, respectively, and should be chosen such that the FTCS stability condition is satisfied. After the inner loop completes, the time step advances and \(\phi\text{New}\) becomes \(\phi\text{Old}\). Each time step can be stored in another data structure for later viewing or analysis if desired. Figure 4 shows the time evolution of the heat equation based on Activity 4.

**Activity 4: FTCS Test Problem**

1. As a test problem, start with the initial condition

\[
\phi(x,0) = \sin\left(\frac{\pi x}{L}\right),
\]

with boundary conditions \(x_0 = x_L = 0\) K. Use the parameters \(L = 1\) m, \(\Delta t = 5 \times 10^{-5}\) s, \(\Delta x = 10^{-2}\) m, and \(t\text{max} = 0.5\) s. This set will give \(r \approx 0.49\), which is below the stability threshold.

2. Create a plot showing the approximate solution at different times up to \(t\text{max}\).

3. The exact solution to Equation 1 with the boundary and initial conditions above is given by
Confirm that the FTCS solution at $t_{\text{max}}$ is approximately the same as the exact solution.

4. Choose values for $\Delta t$, $\Delta x$, and $\alpha$ that give a value of $r > 0.5$. Create the same plot as above. Depending on the chosen parameters, the instability in the FTCS scheme will be seen as an oscillation in the solution curve.

![Figure 4. Time evolution of the heat equation starting from an initial condition $\phi(x,0) = \sin(\pi x/L)$ with parameters $L = 1$ m, $\alpha = 1$ m$^2$/s, and boundary conditions $x_0 = x_L = 0$ K.](image)

**Parallelization**

The OpenMP parallelization of the FTCS algorithm is deceptively tricky. The outermost loop represents the time steps, from the initial condition to the final time specified in $t_{\text{max}}$. Generally, when using OpenMP, we try to parallelize the outer loop of our algorithm so that we capture as much computational work as possible. However, in this case, each time step depends on the previous step. If we distribute the time loop, some threads will not have a correct previous step, because it will not have been computed yet. This situation is known as a loop carried dependency. There can be a variety of solutions to this problem depending on the specific case, but for our problem, we won’t be able to parallelize the outer loop.

The spatial loop, however, provides the independent iterations that we need. On each iteration through the time loop, multiple threads will divide the spatial grid and update their own regions. Overall, this will reduce the amount of computation time required for each time step. With a reasonably large grid size in both time and space, one should achieve good speedup using this parallelization scheme.

**Activity 5: FTCS Parallelization**

1. Use the same test problem as Activity 4. Parallelize the spatial loop using the `#pragma omp` directive. In addition, add calls to `omp_get_wtime()` so that you can compute the elapsed time of the program, specifically the loop over time. Adjust the parameters so that you are below the stability threshold and the calculation takes approximately 10 seconds.

2. Verify that after parallelization you obtain the same results as the serial version.
3. Perform a strong scaling analysis on the code, that is, compute the runtime for a variety of threads, even beyond the number available on your system and plot the execution time versus the number of threads.

The look of the strong scaling analysis will vary depending on the specifications of the system used to do the calculation. As an example, Figure 5 shows the strong scaling on a six-core Intel Xeon processor for up to 12 threads. Most small applications will scale relatively well up to the number of cores on the system. Beyond that, the performance tends to flatten or get worse, mainly due to more threads attempting to use the same limited resources.

Figure 5. Strong scaling analysis of the FTCS solution to the heat equation. Times were computed on a six-core Intel Xeon processor with parameters $\Delta t = 10^{-6}$ s, $\Delta x = 10^{-3}$ m, and $t_{\text{max}} = 0.5$ s. With up to six OpenMP threads, the scaling is nearly exponential. After six threads, the time to solution is slightly longer and there is no additional speedup.

**CONCLUSION**

For a world increasingly dominated by multicore and manycore architectures, learning the basics of parallel programming is important to meet the technological and computational challenges of the future. This article has only scratched the surface of both shared memory programming with OpenMP and numerical solutions to PDEs. There are many more OpenMP directives, clauses, and run time library functions that can help parallelize different types of applications, for example, clauses that can dynamically assign work to threads. The FTCS method for solving the heat equation is a good introduction, but isn’t particularly useful because of the stability condition. For example, the Backward-Time, Centered Space method is unconditionally stable, but must be solved as a system of linear equations. Overall, this case study is a stepping stone to the broader world of computational physics.

**ACKNOWLEDGMENTS**

This work used computing resources provided by the Department of Computer Science and Networking at Wentworth Institute of Technology.

**REFERENCES**


**ABOUT THE AUTHOR**

**Micah D. Schuster** is an assistant professor in the Department of Computer Science and Networking at Wentworth Institute of Technology. Contact him at schusterm@wit.edu.