Data Decomposition

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu

This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License

1D Heat Transfer Example

You have a steel bar. Each section of the bar starts out at a different temperature. There are no incoming heat sources or outgoing heat sinks (i.e., ignore boundary conditions). Ready, go! How do the temperatures change over time?

The fundamental differential equation here is:

\[ \rho C \frac{\partial T}{\partial t} = k \frac{\partial^2 T}{\partial x^2} \]

where:
- \( \rho \) is the density in kg/m
- \( C \) is the specific heat capacity measured in Joules / (kg ∙ °K)
- \( k \) is the coefficient of thermal conductivity measured in Watts / (meter ∙ °K)

In plain words, this all means that temperatures, left to themselves, try to even out. Hots get cooler. Cools get hotter. The greater the temperature differential, the faster the evening-out process goes.

Multicore Block Data Decomposition: 1D Heat Transfer Example

How much the temperature changes in the time step

\[ \Delta T = \frac{k}{\rho C} \left( \frac{\Delta T}{\Delta x^2} \right) \]

How fast the temperature is changing within the bar

\[ \frac{\partial^2 T}{\partial x^2} = \frac{T_{i-1} - 2T_i + T_{i+1}}{(\Delta x)^2} \]

As a side note: the quantity \( k/(\rho C) \) has the unlikely units of m²/sec!

1D Data Decomposition: Partitioning Strategies

On a shared memory multicore system, the obvious approach is to allocate the data as one large global-memory block (i.e., shared).

You will actually need two such arrays, one to hold the current temperature values that you are reading from and one to hold the next temperature values that you are writing to.

1D Data Decomposition: Partitioning

1 2
3 4
5 6

#include <stdio.h>
#include <math.h>
#include <omp.h>
#define NUM_TIME_STEPS          100
#ifndef NUMN
#define NUMN                                  16 // total number of nodes
#endif
#ifndef NUMT
#define NUMT                                     4 // number of threads to use
#endif
#define NUM_NODES_PER_THREAD    ( NUMN / NUMT )
float              Temps[2][NUMN];
int Now;            // which array is the "current values" = 0 or 1
int Next;            // which array is being filled = 1 or 0
void                   DoAllWork( int );
Allocate as One Large Continuous Global Array

```c
omp_set_num_threads(NUMT);
Now = 0;
Next = 1;
for( int i = 0; i < NUMN; i++ )
    Temps[Now][i] = 0.0;
    Temps[Now][NUMN/2] = 100.0;
double time0 = omp_get_wtime();
#pragma omp parallel default(none) shared(Temps,Now,Next)
{
    int me = omp_get_thread_num(  );
    DoAllWork( me ); // each thread calls this
}
double time1 = omp_get_wtime();
double usec = 1000000. * ( time1 - time0 );
double megaNodesPerSecond = (float)NUM_TIME_STEPS * (float)NUMN / usec;
```

Allocate as Separate Thread-Local (private) Sub-arrays

```c
void DoAllWork( int me )
{
    // what range of the global Temps array this thread is responsible for:
    int first = me * NUM_NODES_PER_THREAD;
    int last = first + ( NUM_NODES_PER_THREAD - 1 );
    float left, right;
    float dtemp;
    for( int step = 0; step < NUM_TIME_STEPS; step++ )
    {
        float left = Temps[Now][first-1];
        float dtemp = ( ( K / (RHO*C) ) * ( left - 2.*Temps[Now][first] + Temps[Now][first+1] ) / ( DELTA*DELTA ) ) * DT;
        Temps[Next][first] = Temps[Now][first] + dtemp;
        for( int i = first+1; i <= last-1; i++ )
        {
            float dtemp = ( ( K / (RHO*C) ) * ( Temps[Now][i-1] - 2.*Temps[Now][i] + Temps[Now][i+1] ) / ( DELTA*DELTA ) ) * DT;
            Temps[Next][i] = Temps[Now][i] + dtemp;
        }
    }
    // last element on the right:
    float right = Temps[Now][last+1];
    float dtemp = ( ( K / (RHO*C) ) * ( Temps[Now][last-1] - 2.*Temps[Now][last] + right ) / ( DELTA*DELTA ) ) * DT;
    Temps[Next][last] = Temps[Now][last] + dtemp;
}
```

Allocate as Separate Thread-Global-Heap Sub-arrays

```c
void DoAllWork( int me )
{
    // what range of the global Temps array this thread is responsible for:
    int first = me * NUM_NODES_PER_THREAD;
    int last = first + ( NUM_NODES_PER_THREAD - 1 );
    float left, right;
    float dtemp;
    for( int step = 0; step < NUM_TIME_STEPS; step++ )
    {
        float left = Temps[Now][first-1];
        float dtemp = ( ( K / (RHO*C) ) * ( left - 2.*Temps[Now][first] + Temps[Now][first+1] ) / ( DELTA*DELTA ) ) * DT;
        Temps[Next][first] = Temps[Now][first] + dtemp;
        for( int i = first+1; i <= last-1; i++ )
        {
            float dtemp = ( ( K / (RHO*C) ) * ( Temps[Now][i-1] - 2.*Temps[Now][i] + Temps[Now][i+1] ) / ( DELTA*DELTA ) ) * DT;
            Temps[Next][i] = Temps[Now][i] + dtemp;
        }
    }
    // last element on the right:
    float right = Temps[Now][last+1];
    float dtemp = ( ( K / (RHO*C) ) * ( Temps[Now][last-1] - 2.*Temps[Now][last] + right ) / ( DELTA*DELTA ) ) * DT;
    Temps[Next][last] = Temps[Now][last] + dtemp;
}
```
Allocate as Separate Thread-Global-Heap Sub-arrays

```c
float *nextTemps = new float[NUM_NODES_PER_THREAD];
for(int i = 0; i < NUM_NODES_PER_THREAD; i++)
    nextTemps[i] = Temps[first+i];

// read from Temps[], write into nextTemps[]
for(int steps = 0; steps < NUM_TIME_STEPS; steps++){
    // all the other nodes in between:
    for(int i = 1; i < NUM_NODES_PER_THREAD-1; i++){
        float dtemp = ((K / (RHO*C)) * (Temps[first+i-1] - 2.*Temps[first+i] + Temps[first+i+1]) / (DELTA*DELTA)) * DT;
        nextTemps[i] = Temps[first+i] + dtemp;
    }

    // don't update the global Temps[] until they are no longer being used:
    #pragma omp barrier
    // update the global Temps[]:
    for(int i = 0; i < NUM_NODES_PER_THREAD-1; i++){
        Temps[first+i] = nextTemps[i];
    }

    // be sure all global Temps[] are updated
    #pragma omp barrier
}
```

MegaNodes Computed Per Second

<table>
<thead>
<tr>
<th>Strategy and # of Threads</th>
<th>Number of Nodes</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>One Local Global</td>
<td>8 threads</td>
<td>4 threads</td>
</tr>
<tr>
<td>One Local Global</td>
<td>4 threads</td>
<td>8 threads</td>
</tr>
</tbody>
</table>

Performance as a Function of Number of Nodes

MegaNodes Computed Per Second

<table>
<thead>
<tr>
<th># of Nodes to Compute</th>
<th># of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>2</td>
</tr>
<tr>
<td>200</td>
<td>4</td>
</tr>
<tr>
<td>300</td>
<td>8</td>
</tr>
</tbody>
</table>

Performance as a Function of Number of Threads

MegaNodes Computed Per Second

<table>
<thead>
<tr>
<th># of Threads</th>
<th># of Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>100</td>
</tr>
<tr>
<td>4</td>
<td>200</td>
</tr>
<tr>
<td>8</td>
<td>300</td>
</tr>
</tbody>
</table>

2D Heat Transfer Equation

\[
\frac{\partial T}{\partial t} = \frac{1}{\rho C} \left( \frac{\partial}{\partial x} \left( \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left( \frac{\partial T}{\partial y} \right) \right)
\]

3D Heat Transfer Equation

\[
\frac{\partial T}{\partial t} = \frac{1}{\rho C} \left( \frac{\partial}{\partial x} \left( \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left( \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left( \frac{\partial T}{\partial z} \right) \right)
\]