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

Multicore Block Data Decomposition:
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:

$$\rho C \frac{dT}{dt} = k \frac{\partial^2 T}{\partial x^2}$$

where:

- $\rho$ is the density in kg/m$^3$
- $C$ is the specific heat capacity measured in Joules / (kg $\cdot$ °K)
- $k$ is the coefficient of thermal conductivity measured in Watts / (meter $\cdot$ °K)

= units of Joules/(meter$\cdot$sec$\cdot$°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.

Numerical Methods:
Changing a Derivative into Discrete Arithmetic

How fast the temperature is changing within the bar

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

How much the temperature changes in the time step

$$\frac{\partial T}{\partial t} = \frac{T_t + \Delta t - T_t}{\Delta t}$$

As a side note: the quantity $k/(\rho C)$ has the unlikely units of m$^2$/sec!
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.

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

#define NUM_TIME_STEPS          100

#ifdef NUMN
#define NUMN                                  16 // total number of nodes
#endif

#ifdef 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
int me = omp_get_thread_num();
DoAllWork( me ); // each thread calls this
```

DoAllWork( ), I

```c
void DoAllWork( int me )
{
    // what range of the global Temps array this thread is responsible for:
    int first = me * ( NUM_NODES_PER_THREAD - 1 );
    int last = first + NUM_NODES_PER_THREAD - 1;
    for( int step = 0; step < NUM_TIME_STEPS; step++ )
    {
        // first element on the left:
        {
            float left = 0.;
            if( me != 0 )
                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;
        }
        // all the nodes in between:
        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;
        }
    }
}
```
Because each core is working from left to right across the data, I am guessing that there is little cache line conflict.

We could make each sub-array a thread-local (i.e., private) variable. This would put each sub-array on each thread's individual stack.

The strategy is now to read from the single large global array and compute into each thread's local array.

When we are done, copy each local array into the global array.

We could make each sub-array a thread-heap (also private) variable. This would put each sub-array on the heap.

The strategy is now to read from the single large global array and compute into each thread's heap array.

When we are done, copy each heap array into the global array.
Allocate as Separate Thread-Global-Heap Sub-arrays

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
}
} // for (int steps = 0; …

Performance as a Function of Number of Nodes

Performance as a Function of Number of Threads
2D Heat Transfer Equation

\[ \rho C \frac{\partial T}{\partial t} = k \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} \right) \]

\[ \Delta T = \frac{k}{\rho C} \left( \frac{T_{i,j-1} - 2T_{i,j} + T_{i,j+1}}{(\Delta x)^2} + \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{(\Delta y)^2} \right) \]

3D Heat Transfer Equation

\[ \rho C \frac{\partial T}{\partial t} = k \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right) \]

\[ \Delta T = \frac{k}{\rho C (\Delta x)^2} \left( \frac{T_{i-1,j,k} - 2T_{i,j,k} + T_{i+1,j,k}}{(\Delta x)^2} + \frac{T_{i,j-1,k} - 2T_{i,j,k} + T_{i,j+1,k}}{(\Delta y)^2} + \frac{T_{i,j,k-1} - 2T_{i,j,k} + T_{i,j,k+1}}{(\Delta z)^2} \right) \]