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:

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

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 over time:

\[
\frac{\partial T}{\partial t} = \frac{T_{t+\Delta t} - T_t}{\Delta t}
\]

---

### Multicore Block Data Decomposition:
**1D Heat Transfer Example**

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

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

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

As a side note: the quantity \(k/(\rho C)\) has the unlikely units of \(m^2/\text{sec}\)!
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
#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

Allocate as One Large Continuous Global Array

\[ T_{i-1}, T_i, T_{i+1} \]

Core #0
Core #1
Core #2
Core #3

```c
#pragma omp parallel default(none) shared(Temps,Now,Next)
{
    int me = omp_get_thread_num();
    DoAllWork( me ); // each thread calls this
}
```

double time0 = omp_get_wtime( );

doctor time1 = omp_get_wtime( );

double usec = 1000000. * ( time1 - time0 );

double megaNodesPerSecond = (float)NUM_TIME_STEPS * (float)NUMN / usec;

```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 );
    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;
        }
    }
}
```

What happens if two cores are writing to the same cache line?
False Sharing!
DoAllWork( ), II

```c
// last element on the right:
{
    float right = 0.;
    if( me != NUMT-1 )
        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;
}

// all threads need to wait here so that all Temps[Next][*] values are filled:
#pragma omp barrier

// want just one thread swapping the definitions of Now and Next:
#pragma omp single
{
    Now = Next;
    Next = 1 - Next;
} // implied barrier exists here:
```
Allocate as Separate Thread-Local (private) Sub-arrays

float nextTemps[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; ...  

Allocate as Separate Thread-Global-Heap Sub-arrays

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

```cpp
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>Number of Nodes</th>
<th>Storage Strategy and # of Threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000</td>
<td>One Local 8</td>
</tr>
<tr>
<td>200000</td>
<td>One Local 4</td>
</tr>
<tr>
<td>400000</td>
<td>One Global 8</td>
</tr>
<tr>
<td>600000</td>
<td>One Global 4</td>
</tr>
<tr>
<td>800000</td>
<td>Multithreaded Local Global</td>
</tr>
<tr>
<td>1000000</td>
<td>Multithreaded Global Multithreaded</td>
</tr>
</tbody>
</table>

---

Computer Graphics
Performance as a Function of Number of Nodes

Number of Nodes to Compute vs. MegaNodes Computed Per Second

Performance as a Function of Number of Threads

Number of Threads vs. MegaNodes Computed Per Second
**2D Heat Transfer Equation**

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

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

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

**3D Heat Transfer Equation**

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

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

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