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:

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

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

\( T \) units of Joules/(meter∙sec∙°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

\[ \frac{dT}{dt} = \frac{T_{t+\Delta t} - T_t}{\Delta t} \]

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

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

1D Data Decomposition: Partitioning Strategies

On a distributed computer system, you would have to allocate the data as separate local arrays, each dedicated to its own system.

On a shared memory system, you could 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
#define NUMTRIES                 20
#ifndef NUMN
#define NUMN                    8192
#endif
#ifndef NUMT
#define NUMT                       1
#endif
#define NUM_NODES_PER_THREAD    ( NUMN / NUMT )

int Now;            // which array is the "current values"
int Next;          // which array is being filled
ALIGN64 float           Temps[2][NUMN];
void                    DoAllWork( int );
```
Allocate as One Large Continuous Global Array

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

Allocate as Separate Thread-Global-Heap Sub-arrays

Allocate as One Large Continuous Global Array

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

Allocate as Separate Thread-Global-Heap Sub-arrays
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 ];

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

---

1D Compute-to-Communicate Ratio

- Intracore computing
- Intercore communication

Compute : Communicate ratio = \( N : 2 \)

where \( N \) is the number of compute cells per core

In the above drawing, Compute : Communicate is 4 : 2

How do more Cores Interact with the Compute-to-Communicate Ratio?

- In this case, with 4 cores, Compute : Communicate = 4 : 2
- In this case, with 8 cores, Compute : Communicate = 2 : 2

Think of it as a Goldilocks and the Three Bears sort of thing. :-)

Too little Compute : Communicate and you are spending all your time sharing data values across threads and doing too little computing.

Too much Compute : Communicate and you are not spreading out your problem among enough threads to get good parallelism.

It's difficult to find the "sweet spot" without running experiments.

---

Performance as a Function of Number of Nodes

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

$$\rho c \frac{\partial T}{\partial t} = \frac{\partial}{\partial x} \left( k \frac{\partial T}{\partial x} \right) + \frac{\partial}{\partial y} \left( k \frac{\partial T}{\partial y} \right) + \frac{\partial}{\partial z} \left( k \frac{\partial T}{\partial z} \right)$$

$$\Delta T = \frac{1}{\rho c} \left( \frac{T_{i,j}-2T_{i,j} + T_{i,j-1}}{\Delta y} \right) \Delta x$$

2D Domain (Data) Decomposition

In addition to the issues of size of the compute block, you also have issues of direction.

2D Compute-to-Communicate Ratio

Compute : Communicate ratio = $N^2 : 4N = N : 4$

where $N$ is the dimension of compute nodes per core

The 2D Compute : Communicate ratio is sometimes referred to as Area-to-Perimeter
In 3D problems, this is often (but not always) thought of as:

float Array[NZ][NY][NX];

3D Compute-to-Communicate Ratio

Compute : Communicate ratio = N^3 : 6N^2 = N : 6

where N is the dimension of compute nodes per core

In 3D the Compute : Communicate ratio is sometimes referred to as Volume-to-Surface