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 $\cdot$ °K)
- $k$ is the coefficient of thermal conductivity measured in Watts / (meter $\cdot$ °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.

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.

Allocate as One Large Continuous Global Array

#include <stdio.h>
#include <math.h>
#include <omp.h>
#define NUM_TIME_STEPS 100
#define NUMTRIES 20

#ifdef NUMN
#define NUMN 8192
#endif

#ifdef NUMT
#define NUMT 1
#endif

#define NUM_NODES_PER_THREAD (NUMN / NUMT)

// #define ALIGN64 __declspec(align(64))
#define ALIGN64

int Now; // which array is the "current values"
int Next; // which array is being filled

ALIGN64 float Temps[2][NUMN];

void DoAllWork( );

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 = 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;
    }
}

What happens if two cores are writing to the same cache line?
False Sharing!
Allocate as Separate Thread-Local (private) Sub-arrays

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

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

```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
}
```
Performance as a Function of Number of Nodes

![Graph showing performance as a function of number of nodes.](image)

Performance as a Function of Number of Threads

![Graph showing performance as a function of number of threads.](image)

2D Heat Transfer Equation

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

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

\[ \frac{\Delta T}{\Delta t} = \frac{k}{\rho c} \Delta T_{ij} + \Delta T_{ij} \]

2D Domain (Data) Decomposition

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

![Image of 2D domain decomposition.](image)
Direction Issue: Decomposition Order Matters (think cache)

float Array[A][B];

In 2D problems, this is often (but not always) thought of as:

float Array[NY][NX];

2D Compute-to-Communicate Ratio

Intracore computing

Intercore communication

Compute : Communicate ratio = $\frac{N^2}{4N} = \frac{N}{4}$

where N is the dimension of compute nodes per core

The 2D Compute : Communicate ratio is sometimes referred to as Area-to-Perimeter

3D Heat Transfer Equation

$\rho C_p \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} = k \left( \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \right)

3D Domain (Data) Decomposition
In 3D problems, this is often (but not always) thought of as:

float Array[N][NY][NX];

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*