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)
- \(k/\rho C\) has the unlikely units of m\(^2\)/sec!

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.

How much the temperature changes over time

How fast the temperature is changing within the bar

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.

Allocate as One Large Continuous Global Array

/* Define the number of nodes and the number of threads */
#define NUM_TIME_STEPS          100
#define NUMN                    16 // total number of nodes
#define NUMT                       4 // number of threads to use
#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 );

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:
    if( me == 0 )
    {
      float left = Temps[Now][first-1];
      float dtemp = ( ( K / (RHO*C) ) * ( left - 2.*Temps[Now][first] +  Temps[Now][first+1] ) / ( DELTA*DELTA ) ) * DT;
    }
  }
  // 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;
  }
  // last element on the right:
  if( me != NUMT - 1 )
  {
    float right = Temps[Next][last+1];
  }
  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;
  std::cout << "MegaNodesPerSecond = " << megaNodesPerSecond << std::endl;
DoAllWork( ), II

float right = 0.;
if( me != NUMT-1 )
    right = Temps[Now][last+1];
float dtemp = ( ( K / (RHO*C) ) * \( \frac{\text{Temps[Now][last-1] - 2.*Temps[Now][last] + right}}{\text{DELTA*DELTA}} \) ) * 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:
}

Because each core is working from left to right across the data, I am guessing that there is little cache line conflict.

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

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++)
{
    if (steps > 0)
        Temps = nextTemps;

    // all the other nodes in between:
    for (int i = 1; i < NUM_THREADS - 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
}
```

---

### 1D Compute-to-Communicate Ratio

**Intracore** computing  
**Intercore** communication

**Compute : Communicate ratio = N : 2**  
where N is the number of compute cells per core

---

**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

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

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

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

2D Domain (Data) Decomposition

In addition to the issues of size of the compute block, you also have issues of direction.
**Direction Issue: Decomposition Order Matters (think cache)**

```c
float Array[A][B];
```

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

```c
float Array[NY][NX];
```

**2D Compute-to-Communicate Ratio**

Compute : Communicate ratio = \( N^2 : 4N = N : 4 \)

where \( N \) is the dimension of compute nodes per core

**3D Heat Transfer Equation**

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

**3D Domain (Data) Decomposition**

- 3D Block, *, *
- 3D *, Block, *
- 3D *,*, Block
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