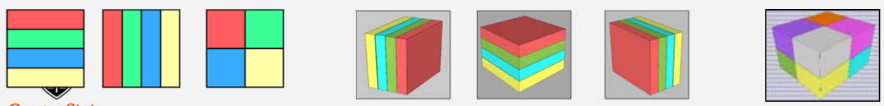
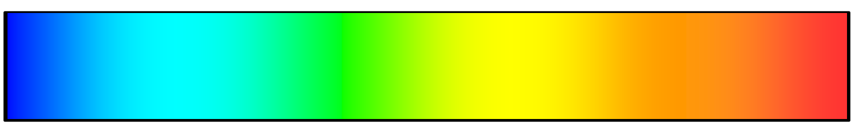
Data Decomposition

Oregon State University
Mike Bailey
mjb@cs.oregonstate.edu



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:
ρ 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)
= 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.

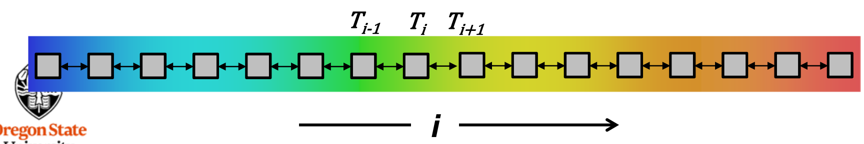
Numerical Methods: Changing a Derivative into Discrete Arithmetic

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

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



Oregon State University Computer Graphics

Multicore Block Data Decomposition: 1D Heat Transfer Example

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

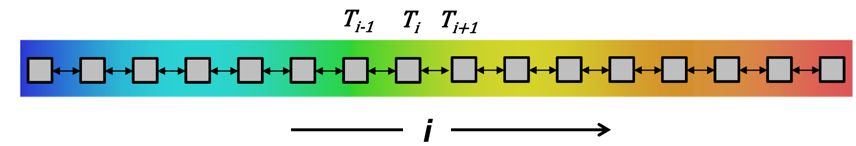
$$\downarrow$$

$$\frac{\Delta T}{\Delta t} = \frac{k}{\rho C}\left(\frac{\Delta^2 T}{\Delta x^2}\right) \longrightarrow \Delta T_i = \left(\frac{k}{\rho C}\right)\left(\frac{T_{i-1} - 2T_i + T_{i+1}}{(\Delta x)^2}\right)\Delta t$$

How much the temperature changes in the time step

Physical properties of the material

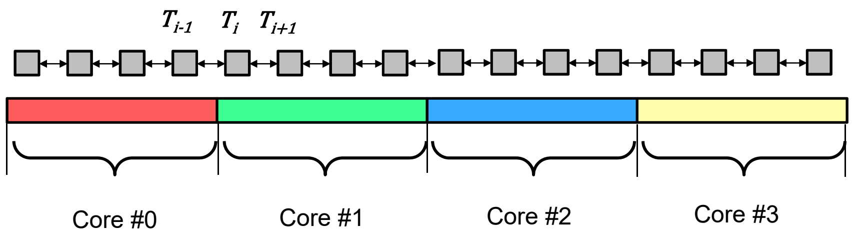
How fast the temperature is changing within the bar



Oregon State University Computer Graphics

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

1D Data Decomposition: Partitioning Strategies



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.

Oregon State University
Computer Graphics

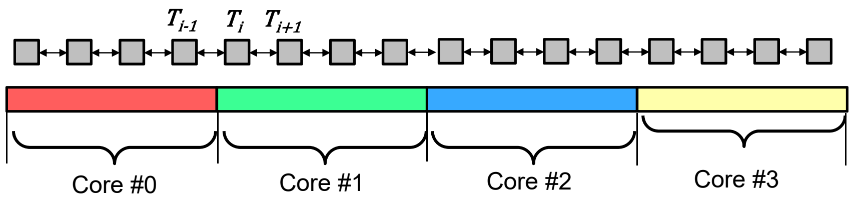
1D Data Decomposition: Partitioning

```
#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 );
```

Oregon State University
Computer Graphics

Allocate as One Large Continuous Global Array

7




```
omp_set_num_threads( NUMT );
Now  = 0;
Next  = 1;

for( int i = 0; i < NUMN; i++ )
        Temps[Now][ i ] =   0.;
Temps[Now][NUMN/2] = 100.;

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;
```


DoAllWork( ), I

8

```
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!

Oregon State University
Computer Graphics

DoAllWork( ), II

9

```
            // 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:

        }  // for( int step = …
}
```

What happens if two cores are writing to the same cache line?
False Sharing!

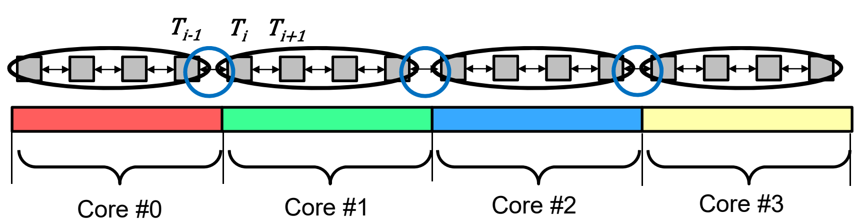
Because each core is working from left to right across the data, I am guessing that there is little cache line conflict.

Oregon State University
Computer Graphics

mjb – March 15, 2020

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

10



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.

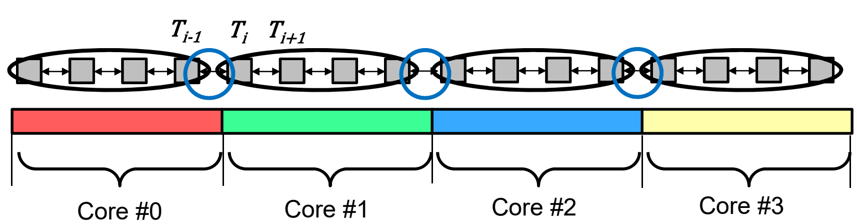
Oregon State University
Computer Graphics

mjb – March 15, 2020

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



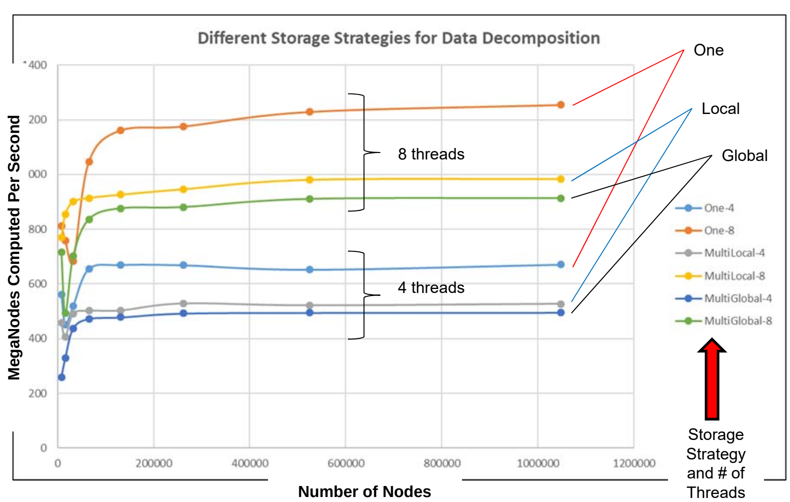
We could make each sub-array a thread-heap (also private) variable. This would put each sub-array on the heap.

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.

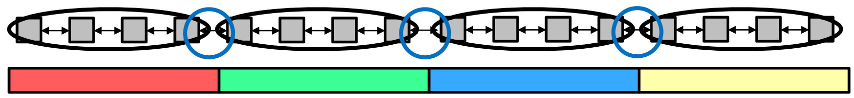
Allocate as Separate Thread-Global-Heap Sub-arrays

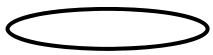
```
float *nextTemps = new float [NUM_NODES_PER_THREAD];
for( int i = 0; i < NUM_NODES_PER_THREAD; i++ )
        nextTemps[ i ] = Temps[first+i];
. . .
// read from Temps[  ],  write into nextTemps[  ]
for( int steps = 0; steps < NUM_TIME_STEPS; steps++ )
{

        // 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; …
```

Oregon State University
Computer Graphics

1D Compute-to-Communicate Ratio





Intracore computing

Intercore communication

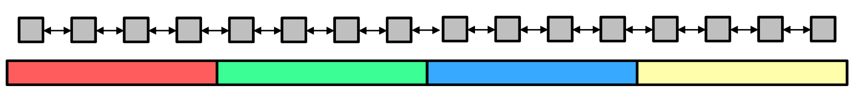
Compute : Communicate ratio = N : 2

where N is the number of compute cells per core

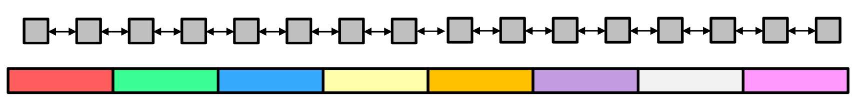
In the above drawing, Compute : Communicate is 4 : 2

Oregon State University
Computer Graphics

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

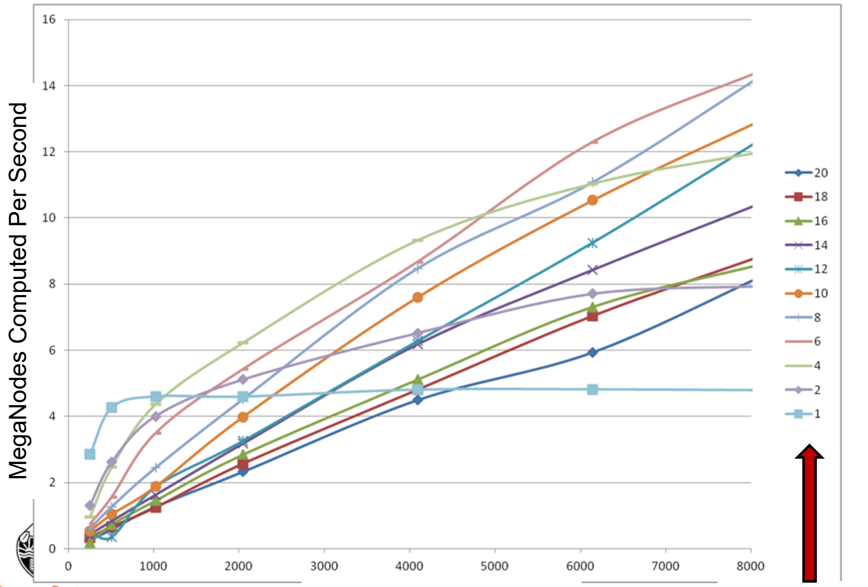
Too little *Compute : Communicate* and you are spending all your time sharing data values across threads and doing too little computing

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

Oregon State University
Computer Graphics

Performance as a Function of Number of Nodes

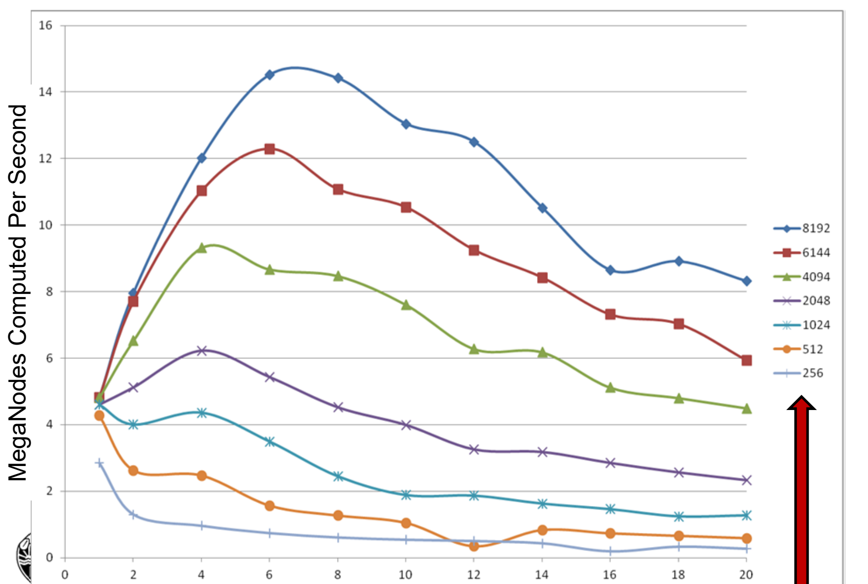


# of Nodes to Compute

# of Threads

Oregon State University Computer Graphics

Performance as a Function of Number of Threads



# of Threads

# of Nodes

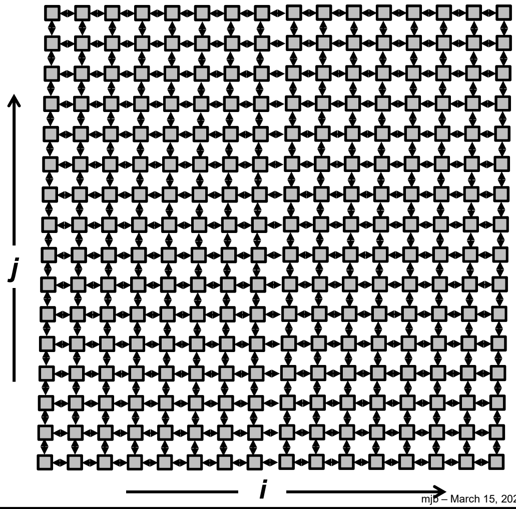
Oregon State University Computer Graphics

2D Heat Transfer Equation

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

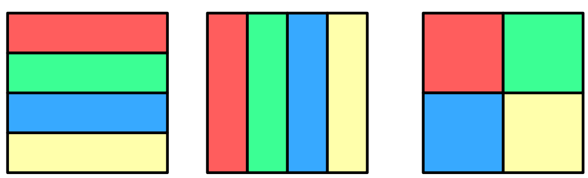
$$\frac{\Delta T}{\Delta t} = \frac{k}{\rho C}\left(\frac{\Delta^2 T}{\Delta x^2} + \frac{\Delta^2 T}{\Delta y^2}\right)$$

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



2D Domain (Data) Decomposition

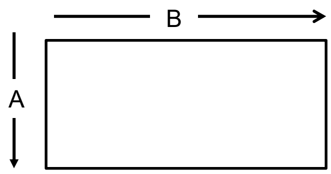
In addition to the issues of size of the compute block, you also have issues of direction.



2D *,Block **2D Block,*** **2D Block, Block**

Direction Issue: Decomposition Order Matters (think cache)

float Array[A][B];

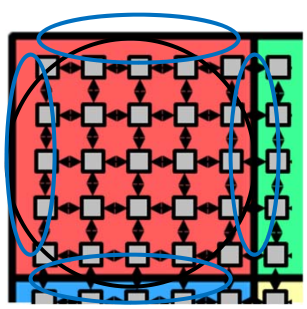


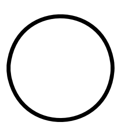
In 2D problems, this is often (but not always) thought of as:

float Array[NY][NX];

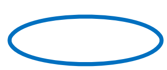
0	0
0	1
0	2
0	3
0	...
0	$B-1$
1	0
1	1
1	2
1	3
1	...
1	$B-1$
.	. .
$A-1$	0
$A-1$	1
$A-1$	2
$A-1$	3
$A-1$	...
$A-1$	$B-1$

2D Compute-to-Communicate Ratio





Intracore computing



Intercore communication

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

where N is the dimension of compute nodes per core

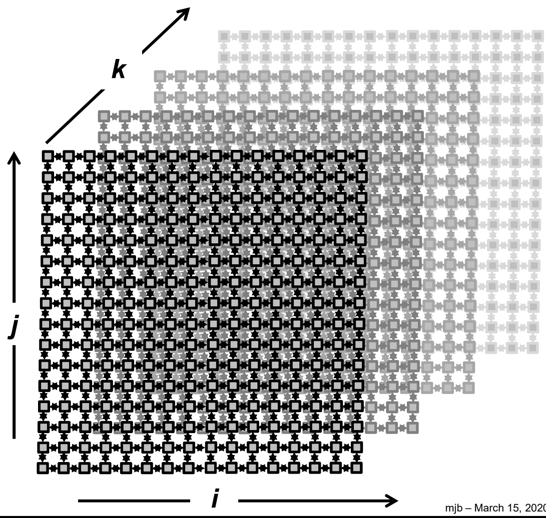
The 2D Compute : Communicate ratio is sometimes referred to as *Area-to-Perimeter*

3D Heat Transfer Equation

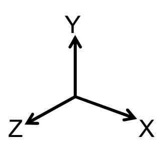
$$\rho C \frac{\partial T}{\partial t} = k\left(\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2}\right)$$

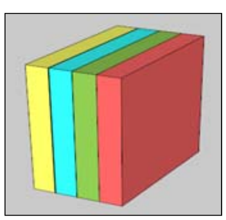
$$\Delta T_{i,j,k} = \left(\frac{k}{\rho C}\right)\left(\frac{T_{i-1,j,k} - 2T_{i,j,k} + T_{i+1,j,k}}{(\Delta x)^2} + \frac{T_{i,j-1,k} - 2T_{i,j,k} + T_{i,j+1,k}}{(\Delta y)^2} + \frac{T_{i,j,k-1} - 2T_{i,j,k} + T_{i,j,k+1}}{(\Delta z)^2}\right)\Delta t$$

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

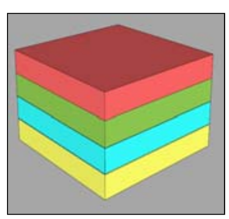


3D Domain (Data) Decomposition

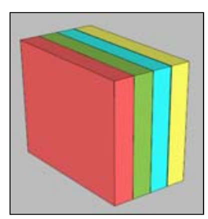




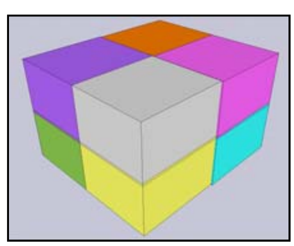
3D Block, *, *



3D *,Block, *



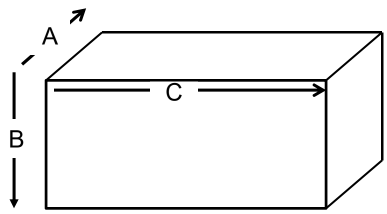
3D *,*,Block



3D Block, Block, Block

Direction Issue: Decomposition Order Matters (think cache)

float Array[A][B][C];



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*