Day 4:
High Performance Computing
CMT106

David W. Walker

Professor of High Performance Computing

Cardiff University

http://www.cardiff.ac.uk/people/view/118172-walker-david

Day 4

- 9:30 – 10:30am: Lecture on application topologies, the vibrating string example code, and its performance analysis.
- 10:30 – 10:50am: Break.
- 10:50am – 12:00pm: Lecture on the Laplace equation example code, and its performance analysis.
- 12:00 – 1:30pm: Lunch break.
- 1:30 – 3:15pm: Lab session, try out the example codes yourself (includes 15min break). Use the Linux lab, or else install MPI on your own system, for example from http://www.mpich.org/.
- 3:15 – 3:45pm: Review of the lab session.
- 3:45pm – 4:00pm: Overview of the second piece of coursework.
- 4:00pm – 5:00pm: Lecture and wrap-up.

Topics Covered on Days 1-4

- *Day 1:* Introduction to parallelism; motivation; types of parallelism; Top500 list; classification of machines; SPMD programs; memory models; shared and distributed memory; OpenMP; example of summing numbers.
- *Day 2*: Interconnection networks; network metrics; classification of parallel algorithms; speedup and efficiency.
- *Day 3*: Scalable algorithms; Amdahl's law; sending and receiving messages; programming with MPI; collective communication; integration example.
- *Day 4: Regular computations and simple examples – the wave equation and Laplace's equation.*

Topics Covered on Days 5-7

- *Day 5*: Programming GPUs with CUDA; CUDA device memory architecture; simple programming examples.
- *Day 6*: Dynamic communication and the molecular dynamics example; irregular computations; the WaTor simulation .
- *Day 7*: Load balancing strategies; message passing libraries; block-cyclic data distribution.

Application Topologies

- In many applications, processes are arranged with a particular topology, e.g., a regular grid.
- MPI supports general application topologies by a graph in which communicating processes are connected by an arc.
- MPI also provides explicit support for Cartesian grid topologies. Mostly this involves mapping between a process rank and a position in the topology.

Cartesian Application Topologies

int MPI_Cart_create (MPI_Comm comm_old, int ndims, int *dims, int *period, int reorder, MPI_Comm *comm_cart)

- Periodicity in each grid direction may be specified.
- Inquiry routines transform between rank in group and location in topology
- For Cartesian topologies, row-major ordering is used for processes, i.e., (i,j) means row i, column j.

Topological Inquiries

- Can get information about a Cartesian topology:

  int MPI_Cart_get(MPI_Comm comm, int maxdims,
  int *dims, int *periods, int *coords)

  This gives information about a Cartesian topology:

  int *dims; // number of processes in each dimension

  int *periods; // periodicity of each dimension

  int *coords; // coordinates of calling process

Mapping Between Rank and Position

- The rank of a process at a given location:

  int MPI_Cart_rank(MPI_Comm comm, int *coords, int *rank)

- The location of a process of a given rank:

  int MPI_Cart_coords(MPI_Comm comm, int rank, int maxdims, int *coords)

Uses of Topologies

- Knowledge of application topology can be used to efficiently assign processes to processors.
- Cartesian grids can be divided into hyperplanes by removing specified dimensions.
- MPI provides support for shifting data along a specified dimension of a Cartesian grid.
- MPI provides support for performing collective communication operations along a specified grid direction.

Topologies and Data Shifts

Consider the following two types of shift for a group of N processes:

- Circular shift by J. Data in process K is sent to process mod((J+K),N)
- End-off shift by J. Data in process K is sent to process J+K if this is between 0 and N-1. Otherwise, no data are sent.

Topologies and Data Shifts 2

- Topological shifts are performed using

  int MPI_Sendrecv(…)

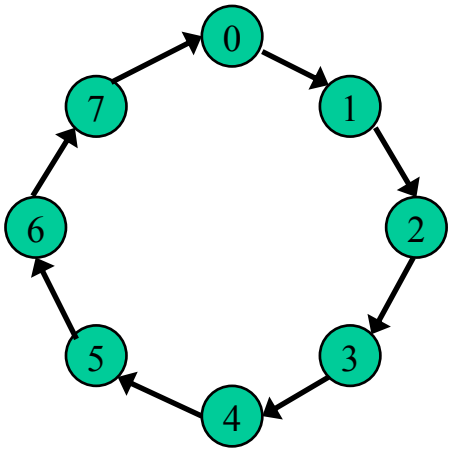
- The ranks of the processes that a process must send to and receive from when performing a shift on a topological group are returned by:

  int MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int *rank_source, int *rank_dest)

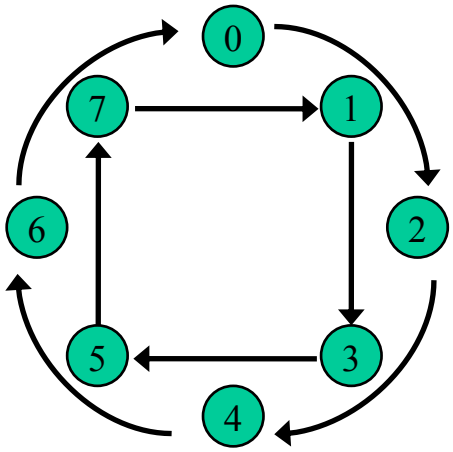
Shifts

Circular:

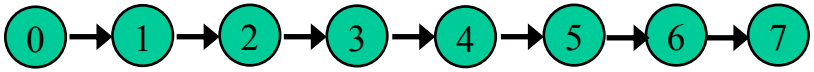
N = 8, J = 1

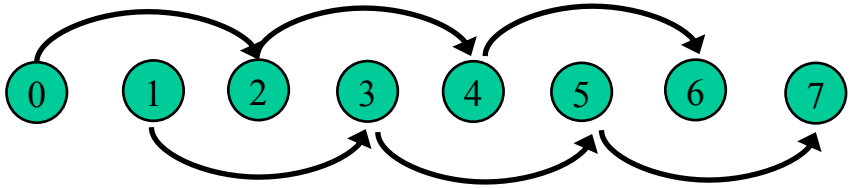


N = 8, J = 2



End-off:





Send/Receive Operations

- In many applications, processes send to one process while receiving from another.
- Deadlock may arise if care is not taken.
- MPI provides routines for such send/receive operations.
- For distinct send/receive buffers:

  int MPI_Sendrecv(…)

- For identical send/receive buffers:

  int MPI_Sendrecv_replace(…)

Vibrating String Problem

We shall now study the vibration of waves on a string, and design a parallel MPI program to solve the partial differential equation that describes the problem mathematically.

$$\frac{\partial^2 \psi}{\partial x^2} = \frac{1}{c^2} \frac{\partial^2 \psi}{\partial t^2}$$

In this equation ψ is the displacement of the string, x is distance along the string, t is time, and c is the wave velocity.

Problem Statement

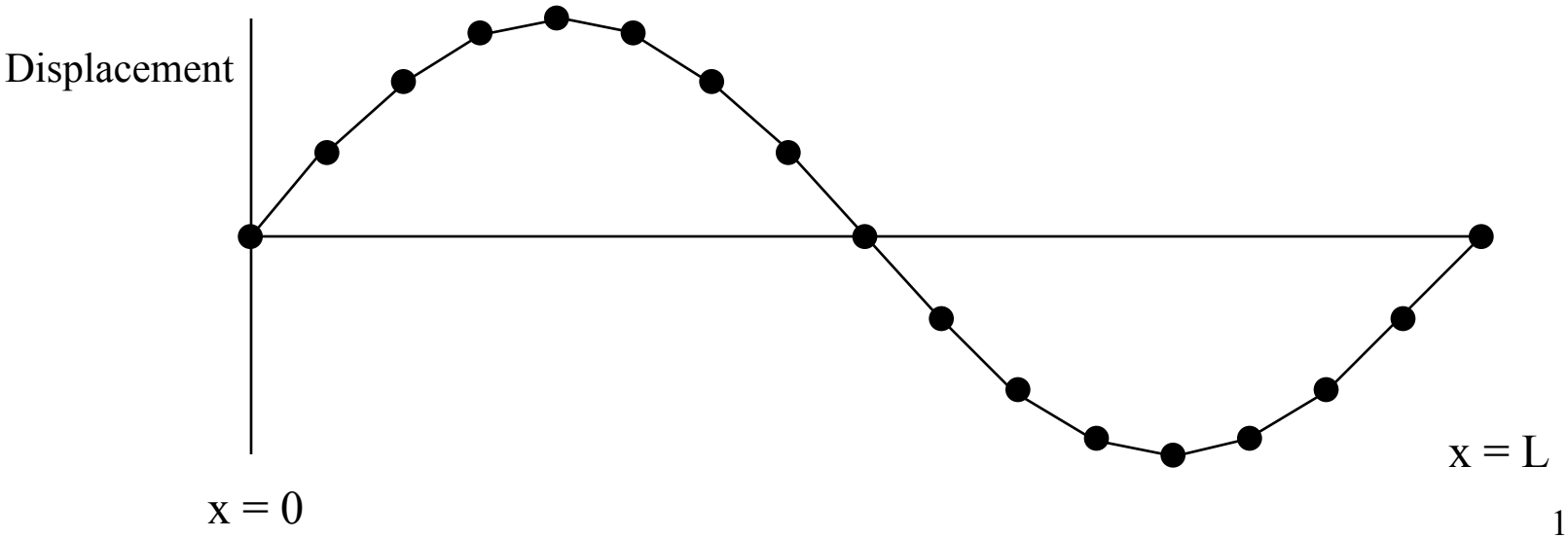
Problem

A string of length L and fixed at each end is initially given a known displacement. What is the displacement at later times?

- Introduce coordinate x so that one end of the string is at x=0 and the other end is at x=L.
- Denote the displacement of the string at position x and time t by $\psi(x,t)$.
- We want to know $\psi(x,t)$.

The Wave Equation

- Mathematically the vibrating string problem is described by the *wave equation*.
- We shall solve this problem numerically by approximating the solution at a number of equally-spaced values of x.



Method of Solution

- We find the solution at a series of time steps, t_0, t_1, t_2, etc.
- At each time step we find the displacement at the points x_0, x_1,…, x_{n-1}, where $x_0=0$ and $x_{n-1}=L$
- At $t_0 = 0$ we assume the string has a known shape, i.e., we know $\psi(x,0)$.
- Given the solution at position x_i at time t_j, the value there at the next time step depends on the current and previous values at that point, and on current values at the neighbouring points.

Numerical Solution

- Approximate partial derivative at x_i and t_j as follows:

$$\frac{\partial \psi}{\partial x} \approx \frac{\psi_{i+1/2,j} - \psi_{i-1/2,j}}{\delta x}$$

- Then:

$$\frac{\partial^2 \psi}{\partial x^2} \approx \frac{(\psi_{i+1,j} - \psi_{i,j}) - (\psi_{i,j} - \psi_{i-1,j})}{\delta x^2} = \frac{\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}}{\delta x^2}$$

$$\frac{\partial^2 \psi}{\partial t^2} \approx \frac{(\psi_{i,j+1} - \psi_{i,j}) - (\psi_{i,j} - \psi_{i,j-1})}{\delta t^2} = \frac{\psi_{i,j+1} - 2\psi_{i,j} + \psi_{i,j-1}}{\delta t^2}$$

$\psi_{i,j}$ is the value of ψ at x_i at time t_j

$\psi_{i+1/2,j}$ is the value of ψ at $x_i + \delta x/2$ at time t_j

Numerical Solution (continued)

- Substitute into the wave equation:

$$\frac{\psi_{i+1,j}-2\psi_{i,j}+\psi_{i-1,j}}{\delta x^2} \approx \frac{1}{c^2}\left(\frac{\psi_{i,j+1}-2\psi_{i,j}+\psi_{i,j-1}}{\delta t^2}\right)$$

- Rearranging gives:

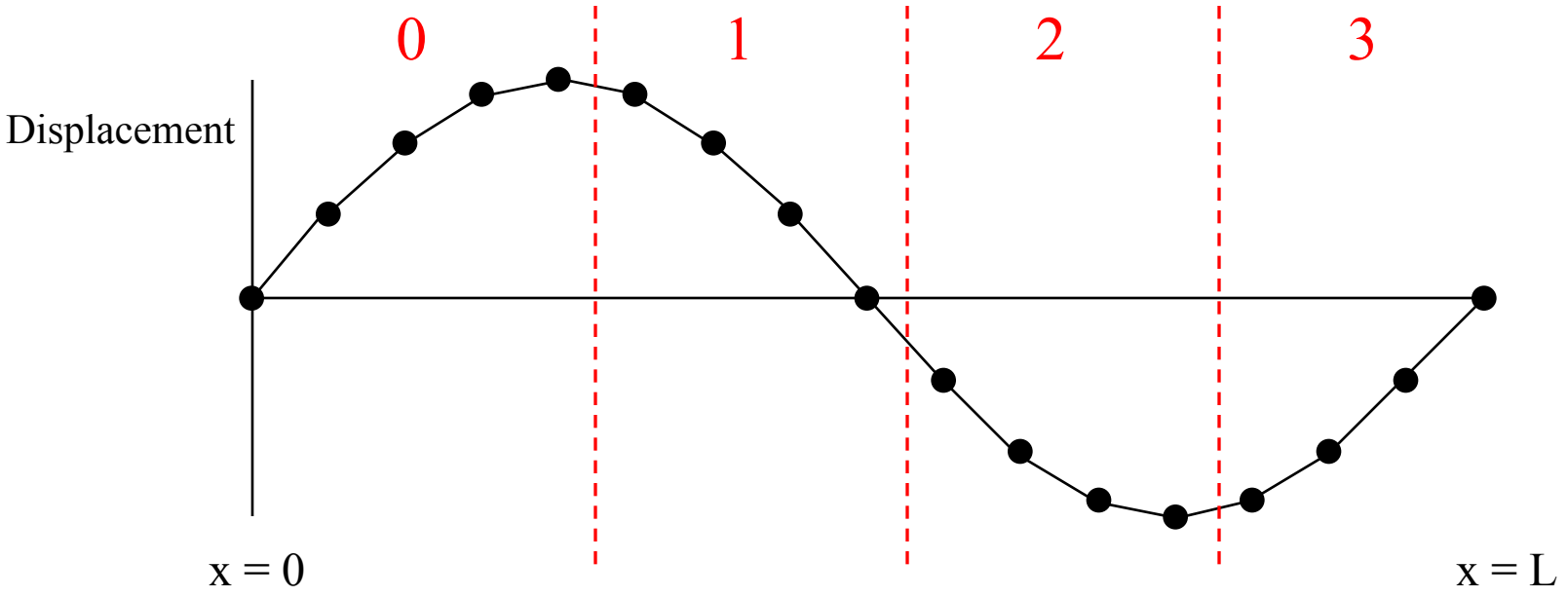
$$\psi_{i,j+1} \approx 2\psi_{i,j} - \psi_{i,j-1} + \tau^2\left(\psi_{i+1,j} - 2\psi_{i,j} + \psi_{i-1,j}\right)$$

where

$$\tau = c\left(\frac{\delta t}{\delta x}\right)$$

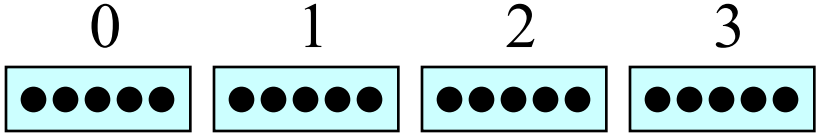
Data Distribution

- Give each process a block of points on the string.
- Each process should have approximately the same number of points to ensure good load balance.



Communication Requirements

- Given the solution at position x_i at time t_j, the value there at the next time step depends on the current and previous values at that point, and on current values at the neighbouring points.
- So to update a point we need to know the displacement at neighbouring points. This entails communication.
- Each process needs to communicate the displacement values for its first and last points before updating its points



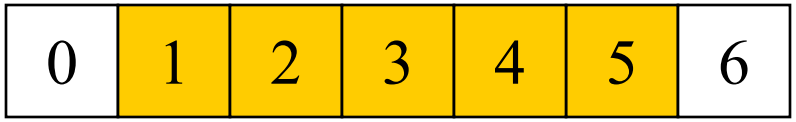
Outline of Parallel Code

- *Initialise data distribution*
  - Find position of each process to determine which block of points it handles.
  - Find out the node numbers of processes to left and right.
- *Initialise arrays*
  - Determine how many points each process handles and the index of the first point in each.
  - Set the psi and oldpsi arrays.
- *Perform Update*
  - Communicate end points.
  - Do update locally.
- *Output results*

Displacement Arrays

- Each process needs to store the endpoint values received from the neighbouring processes. These are stored at the 0 and nlocal+1 positions in the displacement arrays.
- Thus, the displacement arrays need two "extra" entries at each end.

nlocal = 5



Outline MPI Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]){
    MPI_Init (&argc, &argv);

                                        Code for initialising data distribution goes here

                                        Code for initialising arrays goes here

                                        Code for update loop goes here

                                        Code for output phase goes here

    MPI_Finalize();
}
```

Initialising the Data Distribution

```
int rank, nprocs, mypos, periods=0, reorder=1;
MPI_Comm new_comm;
MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);
MPI_Cart_create (MPI_COMM_WORLD, 1, &nprocs, &periods,
                                        reorder,&new_comm);
MPI_Cart_coords (new_comm, rank, 1, &mypos);
MPI_Cart_shift  (new_comm, 0, 1, &left, &right);
```

- new_comm is a new communicator with a 1D Cartesian topology.
- mypos array gives the position of a process in the topology.
- MPI_Cart_shift( ) allows us to find the left and right neighbours of a process.

Initialising the Arrays

```
int nbeg, nend, nlocal, i, alpha, beta;
double *psi, *new_psi, *old_psi, *buf, x, y;

alpha = npts/nprocs;   // npts = α×nprocs + β, 0 ≤ β < nprocs
beta = npts%nprocs;
nlocal = (mypos<beta) ? alpha+1 : alpha;
nbeg = (mypos<beta) ? nlocal*mypos : nlocal*mypos + beta;
nend = nbeg + nlocal - 1;

psi = (double *)malloc(sizeof(double)*(nlocal+2));
new_psi = (double *)malloc(sizeof(double)*(nlocal+2));
old_psi = (double *)malloc(sizeof(double)*(nlocal+2));
buf = (double *)malloc(sizeof(double)*(nlocal+2));

for(i=nbeg;i<=nend;i++){
    x = 2.0*PI*(double)(i)/(double)(npts-1);
    y = sin(x);
    psi[i+1-nbeg] = old_psi[i+1-nbeg] = y;
}
```

Initialising the Arrays

- Let npts = α * nprocs + β.
- If the number of points is exactly divisible by nprocs, then each process has α intervals.
- Otherwise the first β processes have α+1 points, and the rest of the processes have α points.
- nbeg is the index of the first point in each process, i.e., it is the global index corresponding to local index 1.
- We initialise the arrays for indices 1 up to nlocal. Indices 0 and nlocal+1 will be used later to store values received from neighbouring processes.

Update Loop

```
double tau = 0.05;
int istart, iend, tag = 111, nsteps = 500, j;
MPI_Status status;

istart = (mypos==0) ? 2 : 1;
iend   = (mypos==nprocs-1) ? nlocal-1 : nlocal;
for(j=0;j<nsteps;j++){
   MPI_Sendrecv (&psi[1],        1, MPI_DOUBLE, left,  tag,
                 &psi[nlocal+1], 1, MPI_DOUBLE, right, tag,
                 new_comm, &status);
   MPI_Sendrecv (&psi[nlocal], 1, MPI_DOUBLE, right,  tag,
                 &psi[0],      1, MPI_DOUBLE, left,   tag,
                 new_comm, &status);
   for(i=istart;i<=iend;i++)
       new_psi[i] = 2.0*psi[i] - old_psi[i] +
                    tau*tau*(psi[i-1]-2.0*psi[i]+psi[i+1]);
   for(i=1;i<=nlocal;i++){
       old_psi[i] = psi[i];
       psi[i]     = new_psi[i];
   }
}
```

Notes on Update Loop

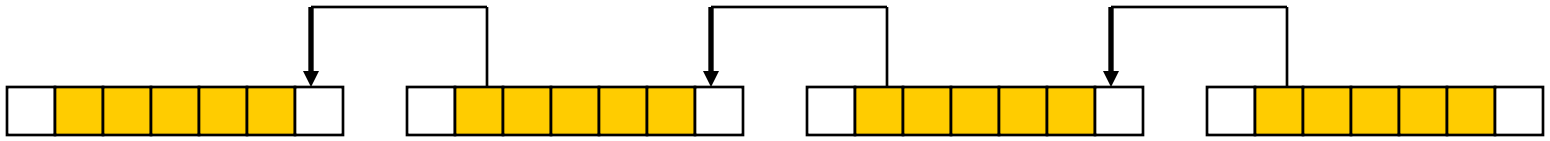
The update phase has 3 main parts:

1. Communicate endpoints between neighbours
2. Update points locally
3. Copy arrays ready for next update step

Communication Code: Left Shift

- All processes send psi[1] to the process to the left, and receive data from the process to the right, storing it in psi[nlocal+1].

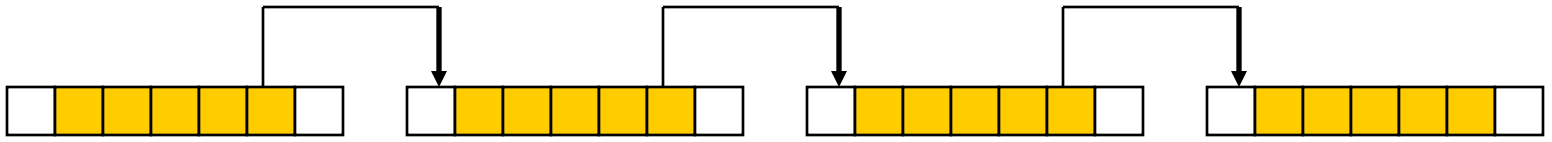
```
MPI_Sendrecv(&psi[1],        1, MPI_DOUBLE, left,  tag,
             &psi[nlocal+1], 1, MPI_DOUBLE, right, tag,
              new_comm, &status);
```



Communication Code: Right Shift

- All processes send psi[nlocal] to the process to the right, and receive data from the process to the left, storing it in psi[0].

```
MPI_Sendrecv (&psi[nlocal], 1, MPI_DOUBLE, right,  tag,
              &psi[0],      1, MPI_DOUBLE, left,   tag,
               new_comm, &status);
```



Output Phase

- We assume the results are output to a file and/or a visualisation device.
- We won't look at this as its mostly a C coding issue.
- One parallel computing issue that arises is whether all processes have access to the file system. Usually they do, but this is not required by MPI.

Performance Analysis

- To analyse the performance of the parallel wave equation code we just look at the update phase.
- To update each point requires 6 floating-point operations in the parallel and sequential codes.
- In the parallel code each process sends and receives two floating-point numbers in each update step.
- We ignore the time to copy to the arrays old_psi and psi.

Performance Analysis 2

The speed-up is:

$$S(N) = \frac{6nt_{calc}}{(6n/N)t_{calc} + 2t_{comm}} = \frac{N}{1 + \tau/(3g)}$$

where N is the number of processes, n is the number of points, $g = n/N$ is the grain size, and $\tau = t_{comm}/t_{calc}$.

Performance Analysis 3

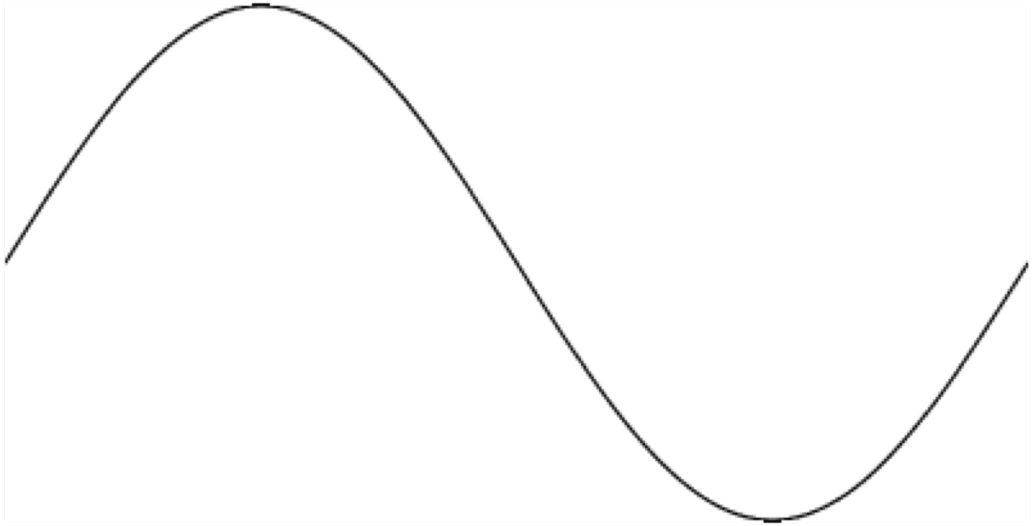
The efficiency is

$$\varepsilon(N) = \frac{1}{1 + \tau/(3g)}$$

so the overhead is $f(N) = \tau/(3g)$.

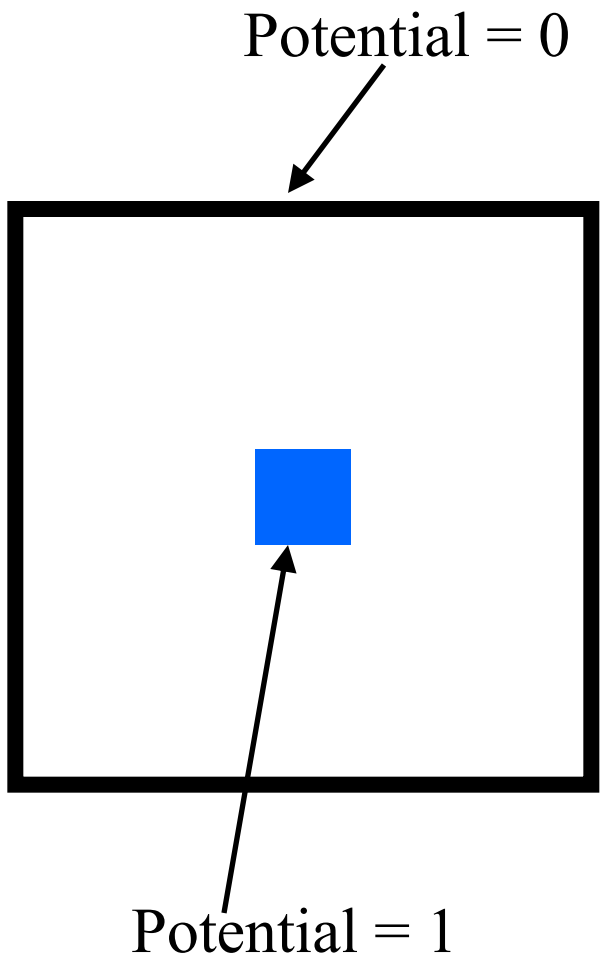
Since the efficiency depends on g but not independently on N the parallel algorithm is perfectly scalable.

Visualizing the Output



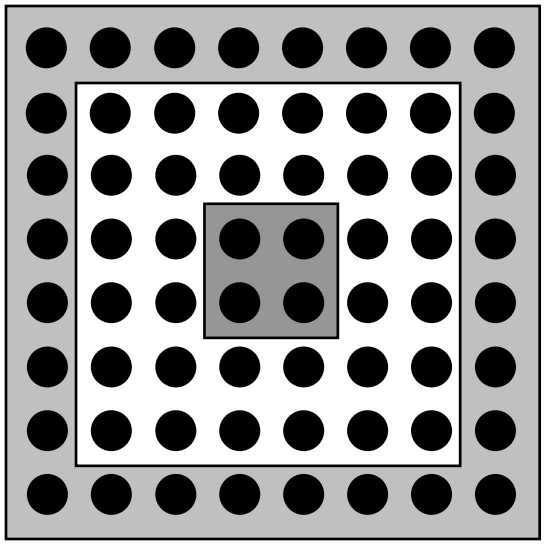
Laplace Equation Problem

- The next problem we shall look at may be used to determine the electric field around a conducting object held at a fixed electrical potential inside a box also at a fixed electrical potential.
- As with the vibrating string problem, this problem can also be expressed mathematically as a partial differential equation, known as the Laplace equation.
- We shall design a parallel MPI program to solve the partial differential equation.



Laplace Equation 2

- This is a 2-D problem whereas the vibrating string was a 1-D problem.
- We divide the problem domain into a regular grid of points, and find an approximation to the solution at each of these points.



- We start with an initial guess at the solution, and perform a series of iterations that get progressively closer to the solution.

Numerical Solution

- The 2D Laplace equation is:

$$\frac{\partial^2 \phi}{\partial x^2} + \frac{\partial^2 \phi}{\partial y^2} = 0$$

- Assume equal spacing, δ, in x and y points:

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{i+1,j} - 2\phi_{i,j} + \phi_{i-1,j}}{\delta^2} \quad \text{and} \quad \frac{\partial^2 \phi}{\partial y^2} \approx \frac{\phi_{i,j+1} - 2\phi_{i,j} + \phi_{i,j-1}}{\delta^2}$$

- Substituting into Laplace equation and re-arranging gives:

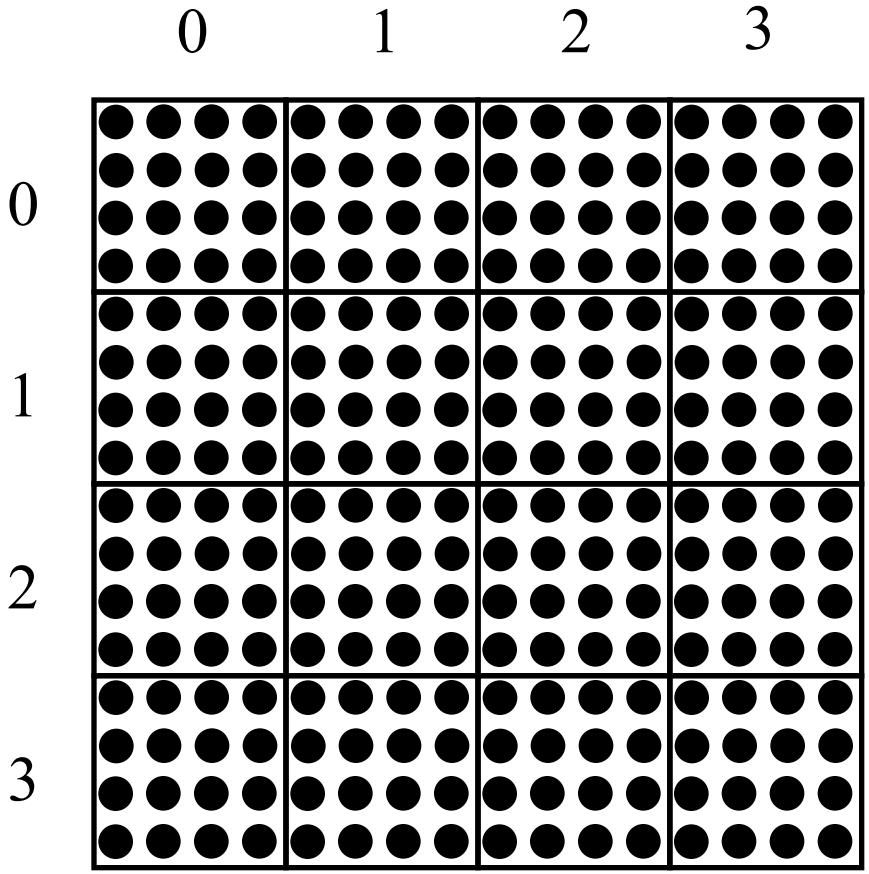
$$\phi_{i,j} = \frac{1}{4}\left(\phi_{i+1,j} + \phi_{i-1,j} + \phi_{i,j+1} + \phi_{i,j-1}\right)$$

$\phi_{i,j}$ is the value of ϕ at position (x_i, y_j)

Data Distribution

- Give each process a 2D block of points.
- Each process should have approximately the same number of points to ensure good load balance.
- Use a MPI's topology routines to map each block of points to a process.

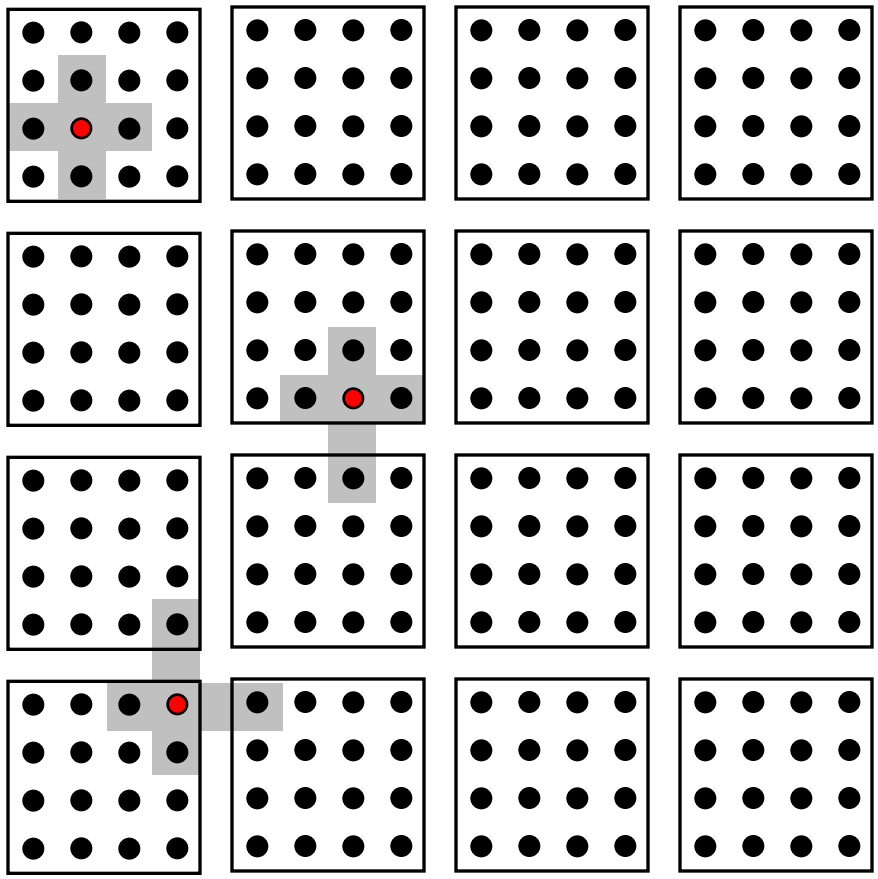
Data Distribution 2



Communication Requirements

- The update formula replaces the solution at a point by the average of the 4 neighbouring points from the previous iteration.
- Points lying along the boundary of a process need data from neighbouring processes.
- Each process needs to communicate the points lying along its boundary before performing an update.

Communication Requirements 2



- To update a red point we need to know the values of the points in the shaded region.
- For points on the edge this requires communication

Outline of Parallel Code

- **Initialise data distribution**
  - Find position of each process to determine which block of points it handles.
  - Find out the node numbers of processes in the left, right, up, and down directions.
- **Initialise arrays**
  - Determine how many points each process handles.
  - Set the phi and mask arrays.
- **Perform update**
  - Copy phi array to oldphi array.
  - Communicate boundary points.
  - Do update locally.
- **Output results**

Array Declarations

- Each process needs to be able to store the boundary values received from its neighbours.
- These are stored in rows 0 and nlocaly+1 and in columns 0 and nlocalx+1 of the phi array.

(5,0)	(5,1)	(5,2)	(5,3)	(5,4)	(5,5)
(4,0)	(4,1)	(4,2)	(4,3)	(4,4)	(4,5)
(3,0)	(3,1)	(3,2)	(3,3)	(3,4)	(3,5)
(2,0)	(2,1)	(2,2)	(2,3)	(2,4)	(2,5)
(1,0)	(1,1)	(1,2)	(1,3)	(1,4)	(1,5)
(0,0)	(0,1)	(0,2)	(0,3)	(0,4)	(0,5)

nlocalx = 4
nlocaly = 4

Array Initialisation

There are 3 arrays:

- phi : the current values of the solution
- oldphi : the values of the solution for the previous iteration.
- mask : equals false on boundaries and true elsewhere.

```
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 1 1 0 0 0
0 0 0 1 1 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

phi

```
F F F F F F F F
F T T T T T T F
F T T T T T T F
F T T F F T T F
F T T F F T T F
F T T T T T T F
F T T T T T T F
F F F F F F F F
```

mask

Outline MPI Code

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <mpi.h>

int main (int argc, char *argv[]) {
    MPI_Init (&argc, &argv);
                                    Code for initialising data
                                    distribution goes here
                                    Code for initialising arrays goes here
                                    Code for update loop goes here
                                    Code for output phase goes here
    MPI_Finalize();
  }
}
```

Initialising the Data Distribution

```
int rank, nprocs, mypos;
int nprocx, nprocy, periods[2], dims[2], coords[2], reorder=1;
MPI_Comm new_comm;

MPI_Comm_size (MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank (MPI_COMM_WORLD, &rank);

dims[0] = dims[1] = 0;
MPI_Dims_create (nprocs, 2, dims);
nprocy = dims[0];
nprocx = dims[1];
periods[0] = periods[1] = 0;
MPI_Cart_create (MPI_COMM_WORLD, 2, dims, periods, 1,&new_comm);
MPI_Cart_coords (new_comm, rank, 2, coords);
myposy = coords[0];
myposx = coords[1];
MPI_Cart_shift  (new_comm, 0, 1, &down, &up);
MPI_Cart_shift  (new_comm, 1, 1, &left, &right);
```

Initialising the Data Distribution 2

- dims[0] and dims[1] are the number of processes in the process grid in each direction. We make the grid as square as possible using MPI_Dims_create().
- This time we set up a 2D communicator, new_comm.
- The MPI_Cart_coords() method gives the position in the topology of each process.
- Calls to MPI_Cart_shift() give the ranks of the neighbouring processes in the four directions.

Allocating the Arrays

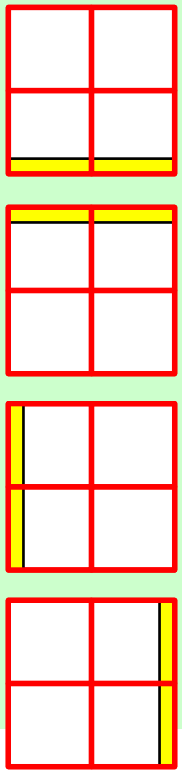
```
int nptsx = 200, nptsy = 200, nsizex, nsizey, bufsize, k;
double *sbuf, *rbuf;
double **phi, **old_phi;
int **mask;

nsizex = (nptsx-1)/nprocx + 1;
nsizey = (nptsy-1)/nprocy + 1;
bufsize = (nsizex>nsizey) ? nsizex : nsizey;
sbuf = (double *)malloc((sizeof(double)*bufsize));
rbuf = (double *)malloc((sizeof(double)*bufsize));
phi  = (double **)malloc((sizeof(double*)*(nsizey+2)));
oldphi = (double **)malloc((sizeof(double*)*(nsizey+2)));
mask = (int **)malloc((sizeof(int*)*(nsizey+2)));
for (k=0;k<nsizey+2;k++){
     phi[k] = (double *)malloc(sizeof(double)*(nsizex+2));
     oldphi[k] = (double *)malloc(sizeof(double)*(nsizex+2));
     mask[k] = (int *)malloc(sizeof(int)*(nsizex+2));
}
```

Initialising phi and mask Arrays

- Set all of phi to 0, and all of mask to true.
- For processes in row 0 of the process mesh we must set row 1 of the mask array to false.
- For processes in the last row of the process mesh we must set row nlocaly of the mask array to false.
- For processes in column 0 of the process mesh we must set column 1 of the mask array to false.
- For processes in the last column of the process mesh we must set column nlocalx of the mask array to false.
- For the 4 points in the centre we must set the phi and mask entries to 1 and false, respectively, in the processes containing them.

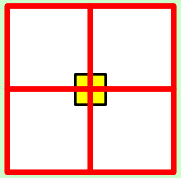
Initialisation of Arrays

```
int i, j, nlocalx, nlocaly;
nlocalx = (myposx==nprocx-1) ? nptsx-nsizex*(nprocx-1) : nsizex;
nlocaly = (myposy==nprocy-1) ? nptsy-nsizey*(nprocy-1) : nsizey;

for(j=0;j<=nlocaly+1;j++)
    for(i=0;i<=nlocalx+1;i++){
       phi[j][i]  = 0.0;
       mask[j][i] = 1;
    }

if (myposy == 0)
    for(i=0;i<=nlocalx+1;i++) mask[1][i] = 0;

if (myposy == nprocy-1)
    for(i=0;i<=nlocalx+1;i++) mask[nlocaly][i] = 0;

if (myposx == 0)
    for(j=0;j<=nlocaly+1;j++) mask[j][1] = 0;

if (myposx == nprocx-1)
    for(j=0;j<=nlocaly+1;j++) mask[j][nlocalx] = 0;
```



Initialisation of Arrays 2

```
 int globalx, globaly;

 for(j=1;j<=nlocaly;j++){
   global_y = nlocaly*myposy + j - 1;
   if (global_y == nptsy/2 || global_y == nptsy/2-1){
      for(i=1;i<=nlocalx;i++){
         global_x = nlocalx*myposx + i - 1;
         if (global_x == nptsx/2 || global_x == nptsx/2-1){
            mask[j][i] = 0;
            phi[j][i]  = 1.0;
         }
      }
   }
}
```



Update Phase

The update phase has three main parts.

- Copy phi to oldphi array.
- Communicate boundary data.
- Update points locally.

Update Phase 2

```
int nsteps = 500;

for(k=1;k<=nsteps;k++){
    for(j=1;j<=nlocaly;j++)
       for(i=1;i<=nlocalx;i++)
          oldphi[j][i] = phi[j][i];;

    for(j=1;j<=nlocaly;j++)
       for(i=1;i<=nlocalx;i++)
          if (mask[j][i]) phi[j][i] = 0.25*(oldphi[j][i-1] +
             oldphi[j][i+1] + oldphi[j-1][i] + oldphi[j+1][i]);
}
```

Shift up

Shift down

Shift right

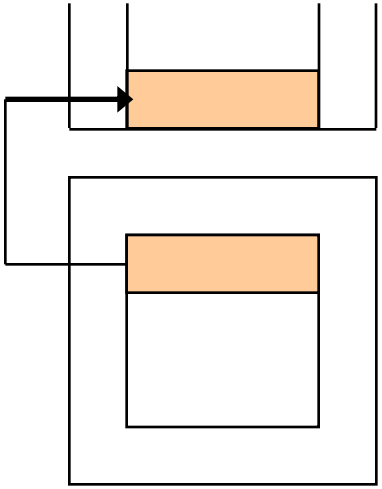
Shift left

Communication

- Communication takes place by shifting data in each of the four directions (left, right, up, and down).
- Before communicating in any direction we must explicitly buffer the data to be sent, and unpack it when it is received.

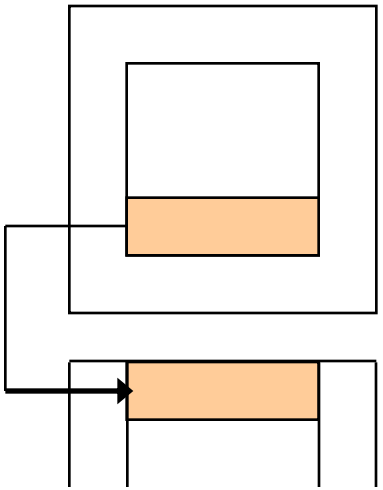
Shift Up

```
MPI_Status status;

MPI_Sendrecv (&oldphi[nlocaly][1],nlocalx,MPI_DOUBLE, up,tag,
                &oldphi[0][1], nlocalx, MPI_DOUBLE,down, tag,
                new_comm, &status);
```



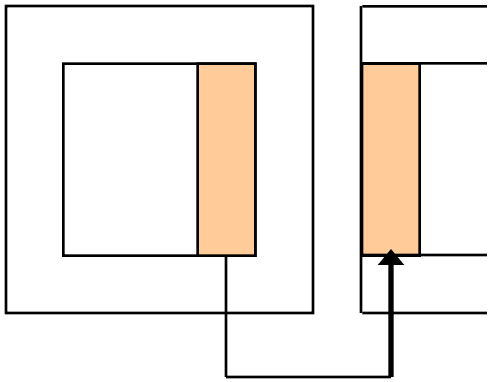
Shift Down

```
MPI_Sendrecv (&oldphi[1][1],  nlocalx, MPI_DOUBLE,  down, tag,
              &oldphi[nlocaly+1][1], nlocalx, MPI_DOUBLE, up, tag,
               new_comm, &status);
```



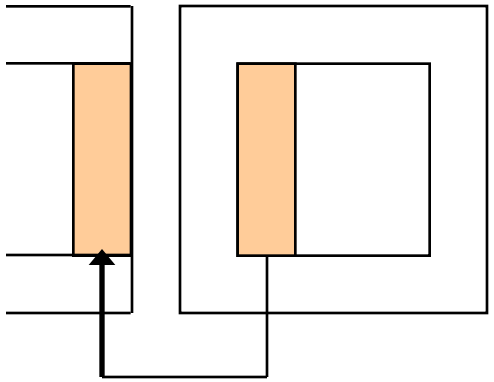
Shift Right

```
for(i=1;i<=nlocaly;i++) sbuf[i-1] = oldphi[i][nlocalx];
MPI_Sendrecv (sbuf, nlocaly, MPI_DOUBLE, right, tag,
              rbuf, nlocaly, MPI_DOUBLE, left,  tag,
               new_comm, &status);
for(i=1;i<=nlocaly;i++) oldphi[i][0] = rbuf[i-1];
```



Shift Left

```
for(i=1;i<=nlocaly;i++) sbuf[i-1] = oldphi[i][1];

MPI_Sendrecv (sbuf, nlocaly, MPI_DOUBLE, left, tag,
              rbuf, nlocaly, MPI_DOUBLE, right,tag,
              new_comm, &status);

for(i=1;i<=nlocaly;i++) oldphi[i][nlocalx+1] = rbuf[i-1];
```



Performance Analysis

- The update formula requires 4 floating-point operations per grid point.
- The number of grid points per processor shifted in the left/right direction is n/P, where n x n is the size of the grid and P is the number of processors in one column of the processor mesh.
- The number of grid points per processor shifted in the up/down direction is n/Q, where Q is the number of processors in one row of the processor mesh.

Speed Up

The speed-up is:

$$S(N) = \frac{4n^2 t_{calc}}{(4n^2/N)t_{calc} + (2n/Q)t_{shift} + (2n/P)t_{shift}}$$

$$= \frac{N}{1 + (P+Q)\,\tau/(2n)}$$

$$= \frac{N}{1 + (Q/n)(1+\alpha)\tau/2}$$

where M=n x n is the size of the grid, P x Q is the processor mesh, $P=\alpha Q$, and $\tau=t_{shift}/t_{calc}$.

Efficiency and Overhead

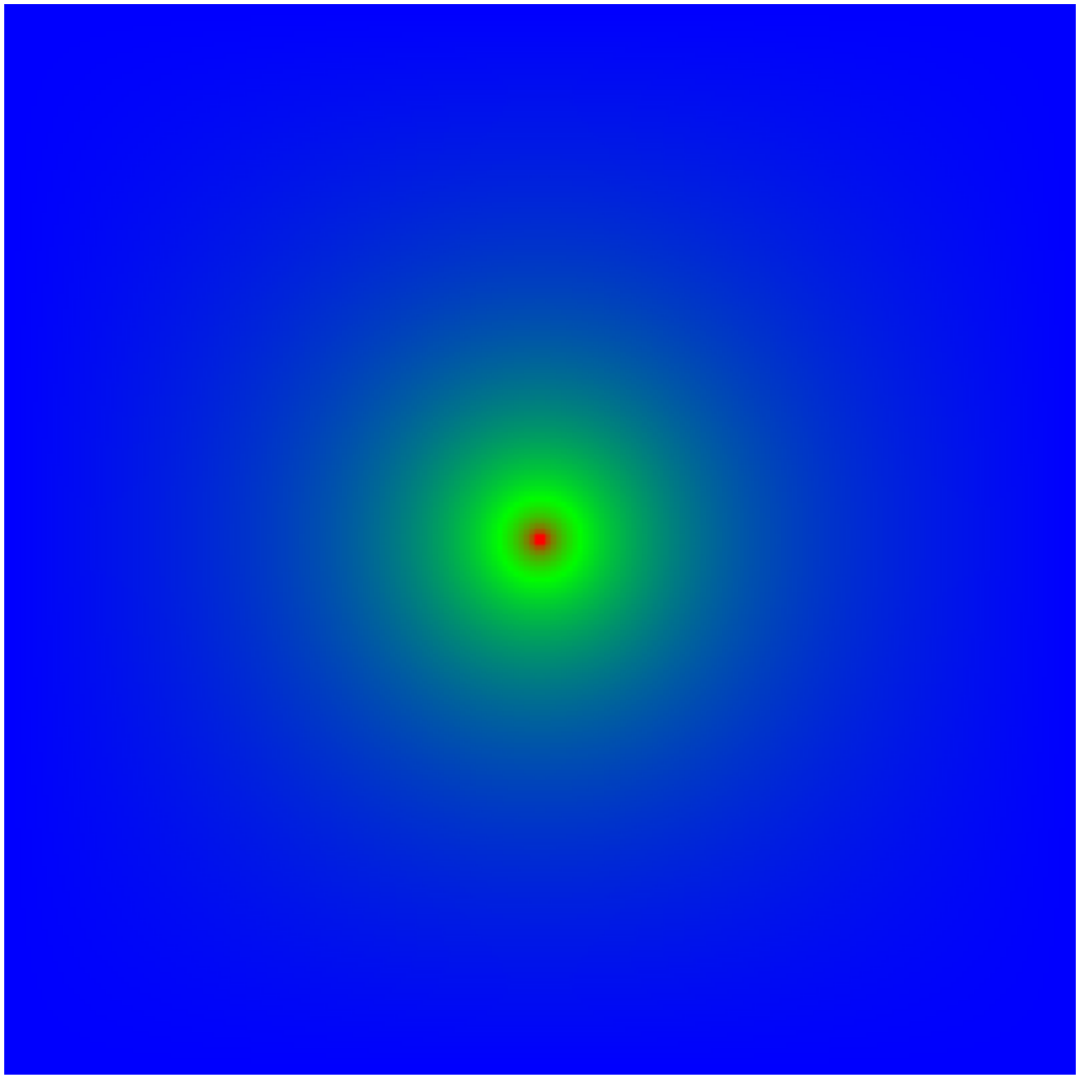
- Since $N=PQ=\alpha Q^2$ and $M=n^2$ is the number of points, the efficiency is given by:

$$\varepsilon(N) = \frac{1}{1 + (1+\alpha)/(2\sqrt{\alpha})(\tau/\sqrt{g})}$$

where $g = M/N$ is the grain size.

- Since the efficiency depends only on g, and not independently on n and N, the algorithm is perfectly scalable.

Visualizing the Output



CUDA

- "Compute Unified Device Architecture" introduced by NVidia in 2007.
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- Targeted software stack
  - Compute oriented drivers, language, and tools
- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute – graphics-free API
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management

Useful CUDA Resources

1. Textbook: “Programming Massively Parallel Processors,” David B. Kirk and Wen-mei W. Hwu, third edition, pub. Morgan Kaufmann, 2016. ISBN 978-0-12-811986-0. https://www.elsevier.com/books/programming-massively-parallel-processors/kirk/978-0-12-811986-0
2. *NVidia CUDA Programming Guide*, available at http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html - this is for v9.0.

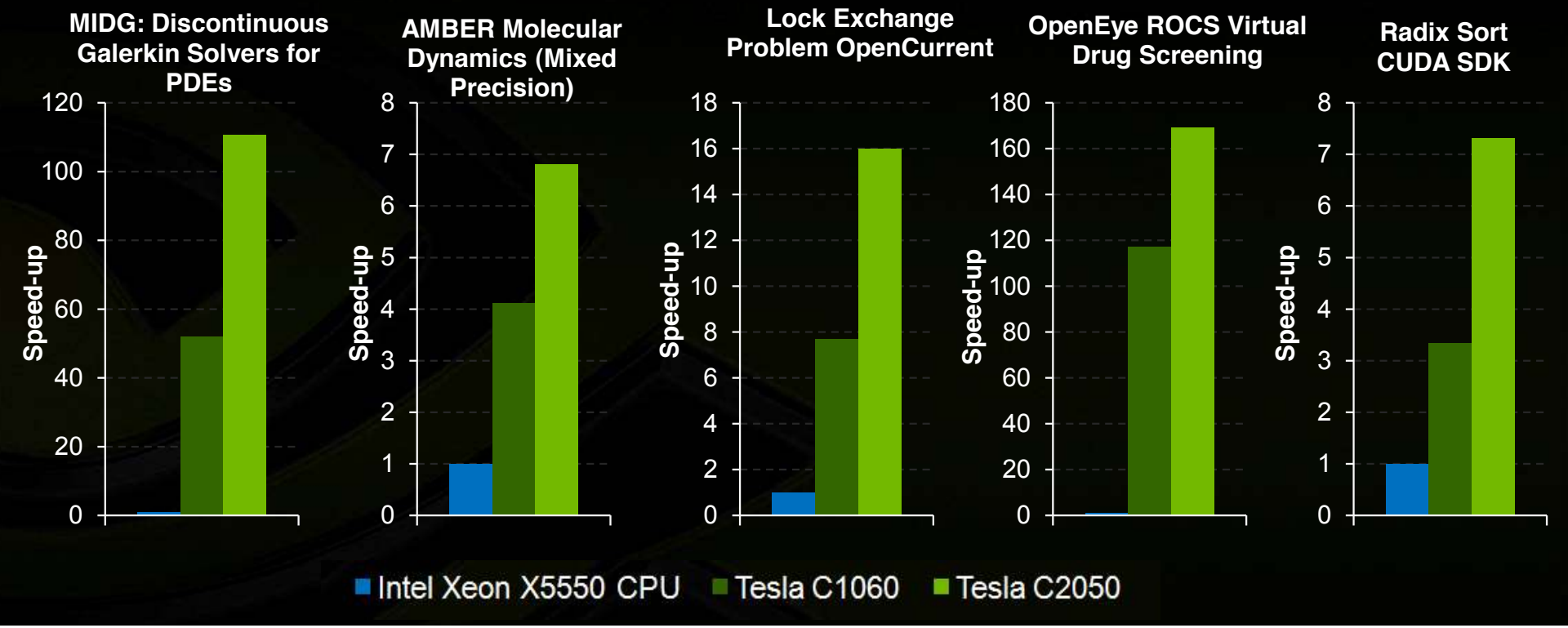
Can You Run CUDA?

- If you have a laptop with an Nvidia GPU, then you may be able to run CUDA on it.
- Find out what GPU your Laptop has and then check if it supports CUDA at:

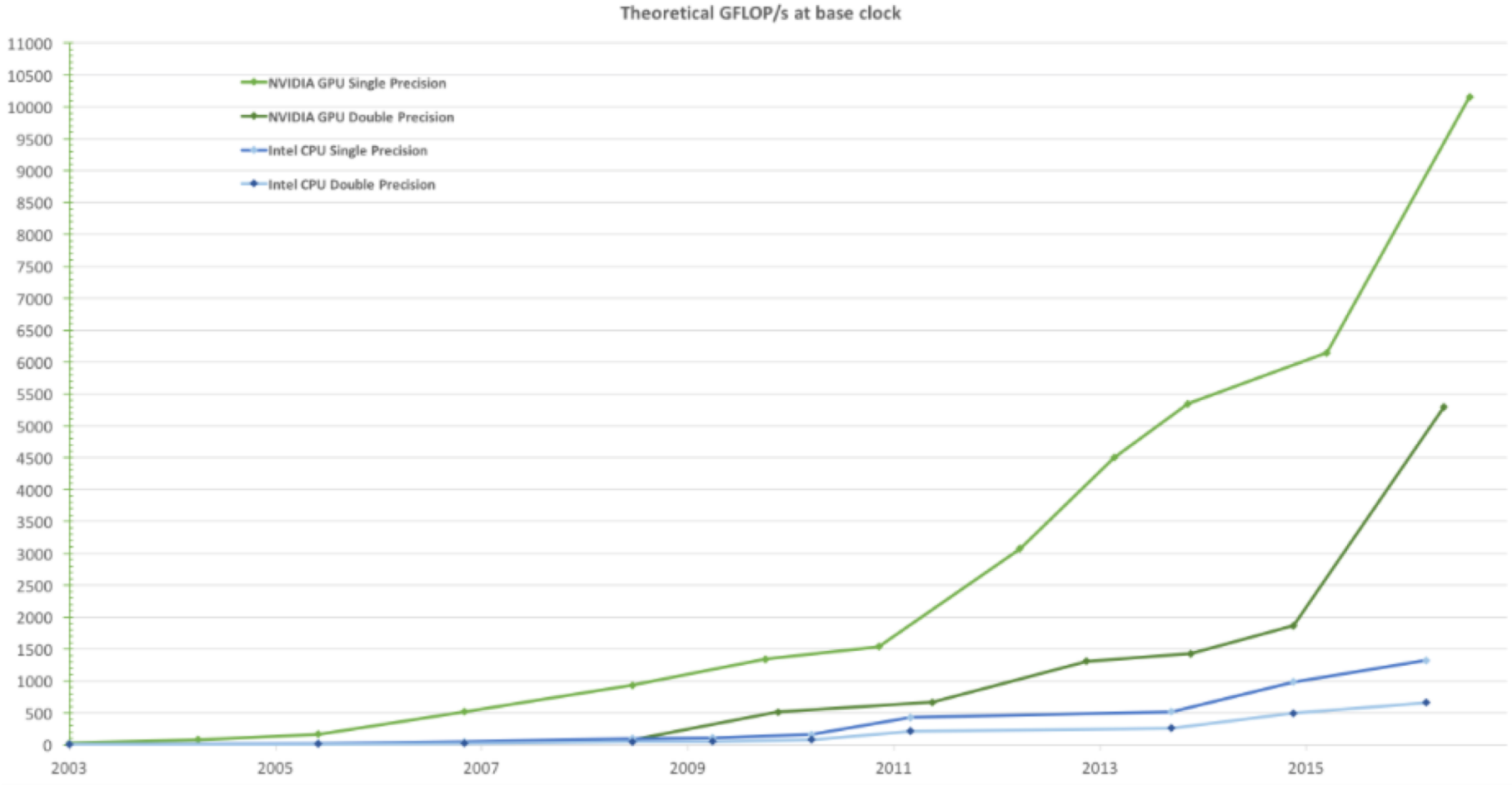
  https://developer.nvidia.com/cuda-gpus
- If you can run CUDA then you can download it from:

  https://developer.nvidia.com/cuda-toolkit

Performance Summary

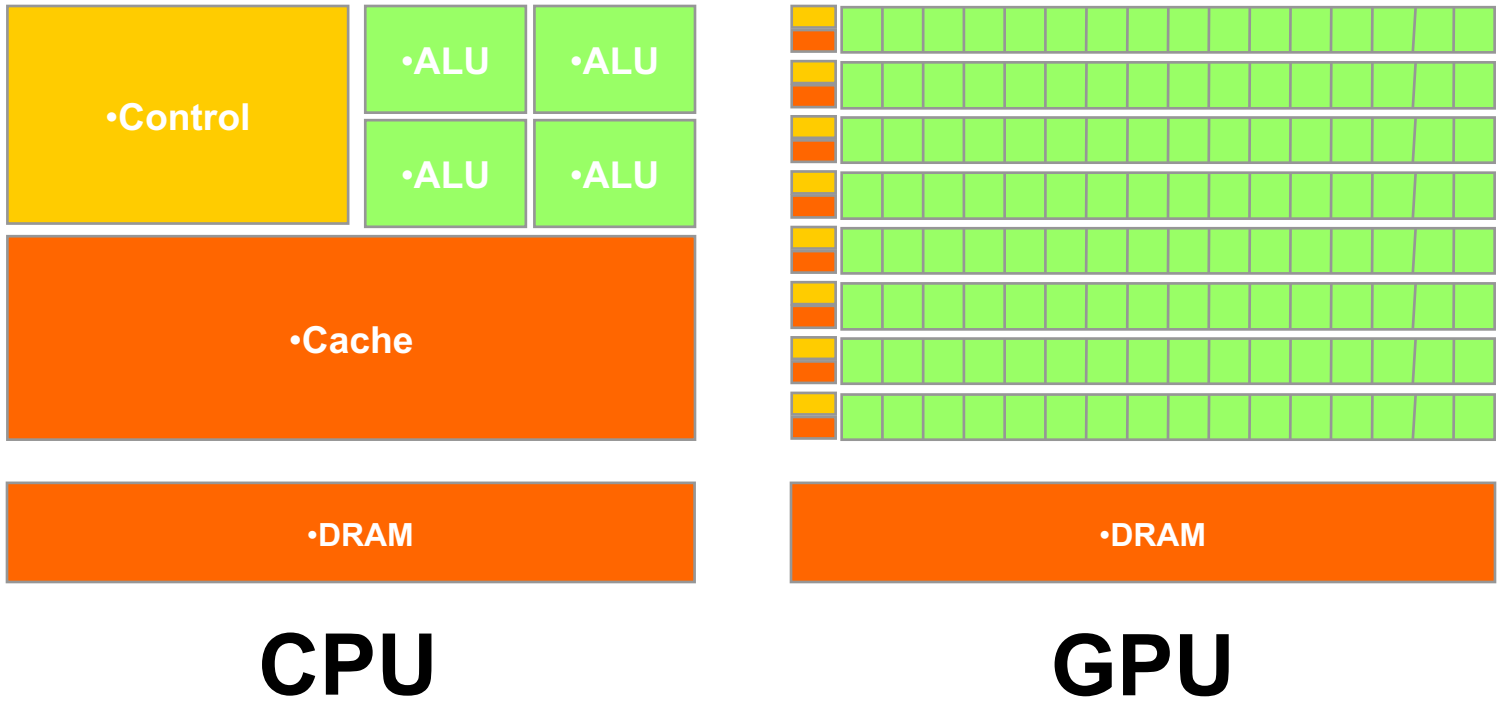


Why Massively Parallel Processing

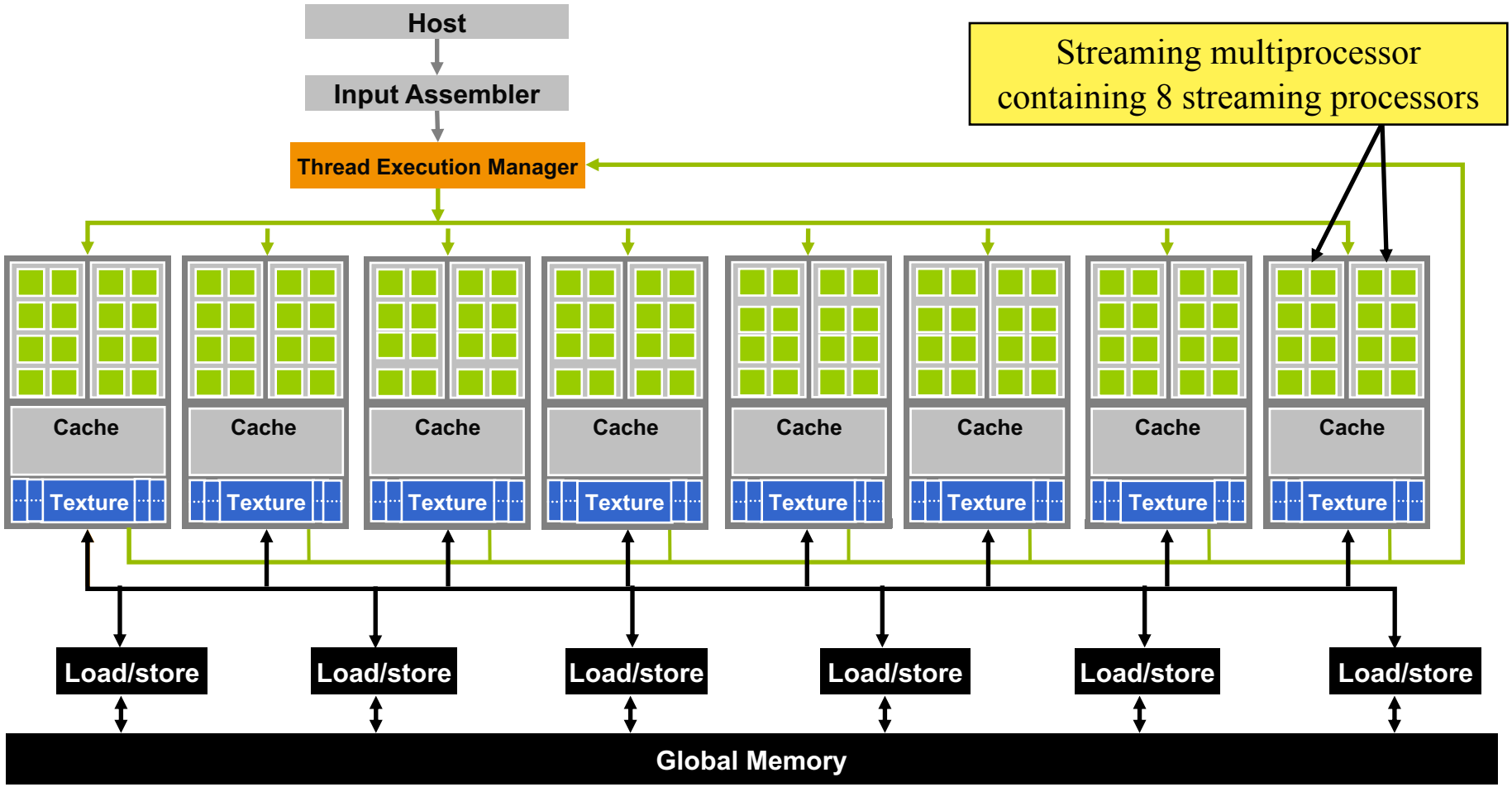


- A quiet revolution and potential build-up
  - Calculation: 1 TFLOPS vs. 32 GFLOPS
  - Memory Bandwidth: 100 GB/s vs. 8.4 GB/s
- GPU in every PC and workstation – massive volume and potential impact

CPUs and GPUs have fundamentally different design philosophies



Architecture of a CUDA-capable GPU

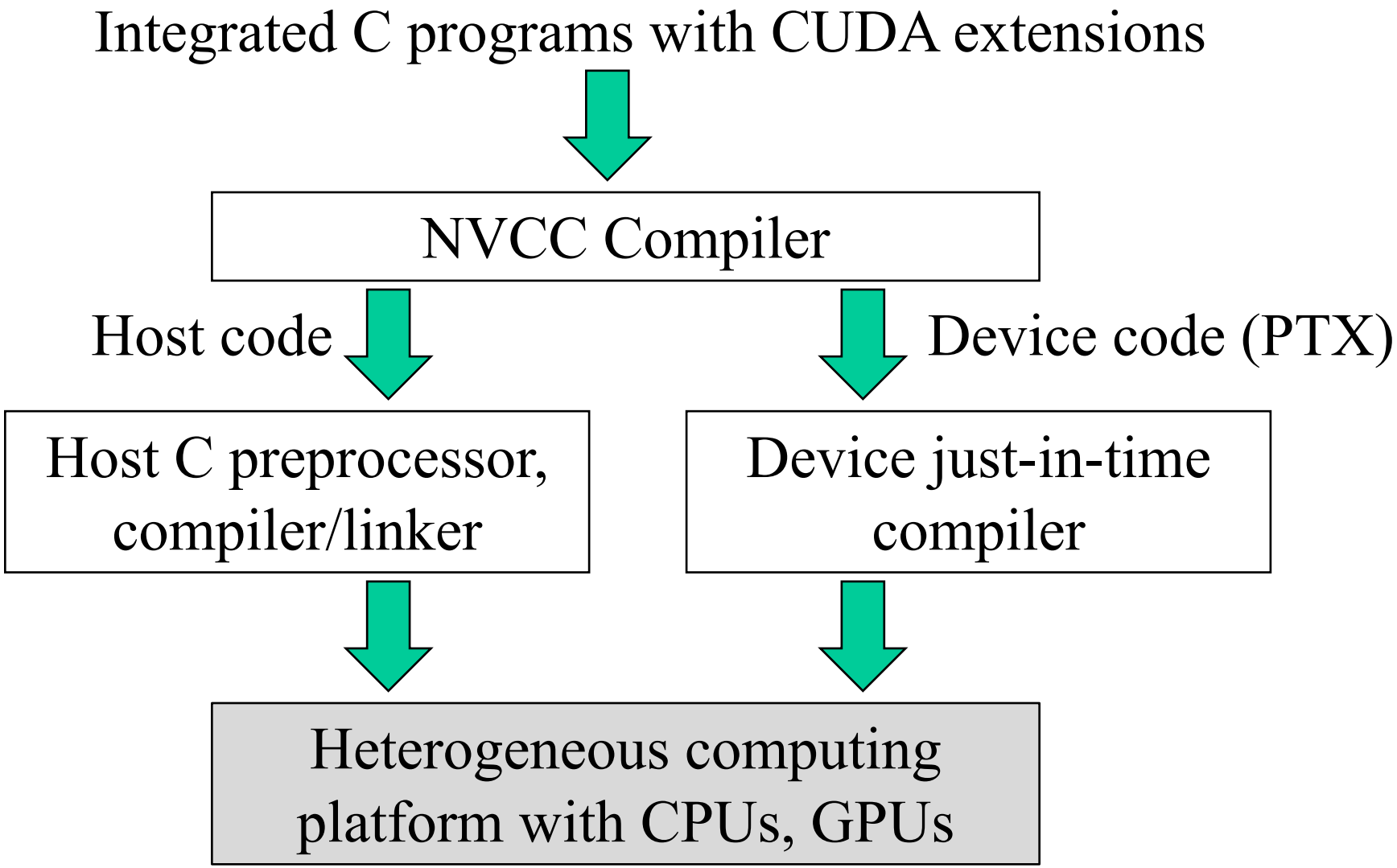


Structure of a CUDA Program

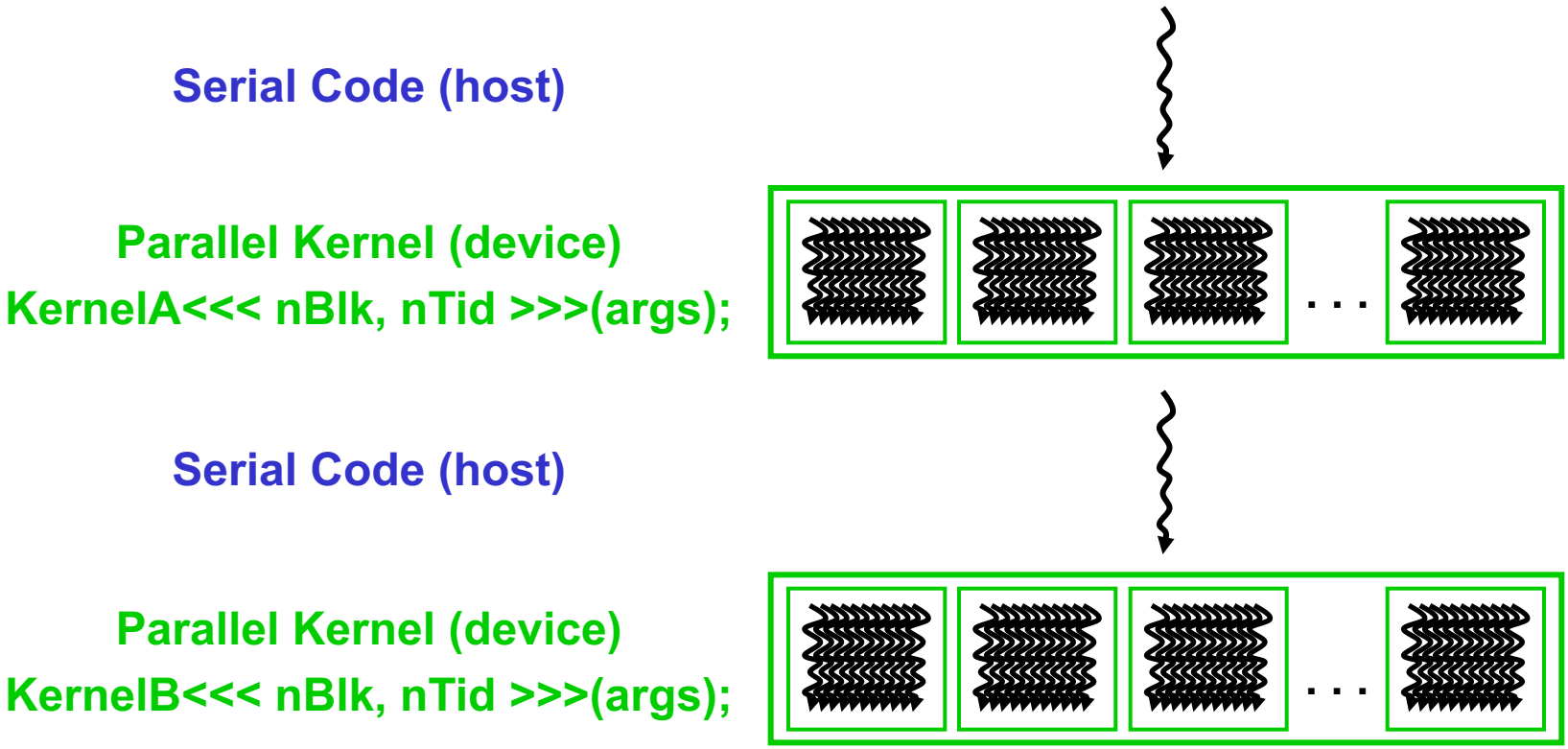
- CUDA programs involve coordination between a *host* (CPU) and one or more *devices* (GPUs).
- A CUDA source file may consist of both host and device code.
- Can add device functions and data declarations to any traditional C code.

Structure of a CUDA Program

- CUDA source files have .cu suffix.
- Device code is marked with CUDA keywords for labelling data-parallel functions called *kernels*.
- Source file is compiled with *nvcc* compiler which gives standard host executable and device code in PTX format.
- PTX is a low-level parallel thread execution virtual machine and instruction set architecture.


Integrated C programs with CUDA extensions
NVCC Compiler
Host code
Device code (PTX)
Host C preprocessor, compiler/linker
Device just-in-time compiler
Heterogeneous computing platform with CPUs, GPUs

Execution of a CUDA Program



- Host launches KernelA and then KernelB on the GPU.
- All the threads generated by a kernel launch are collectively called a *grid*.

Thread Generation

- Launching a kernel typically launches a large number of threads to exploit data parallelism.
- Can assume that these threads take very few clock cycles to generate and schedule due to efficient hardware support.
- This contrasts with traditional CPU threads that typically take thousands of clock cycles to generate and schedule.

Lab Equipment

- We shall use the Linux lab machines in C/2.08.
- Most of these have one NVidia GeForce RTX 2700 GPU with 2 GB of memory.
- Using CUDA 10.0
- Compile with nvcc:

  nvcc –o code code.cu

Device Properties

- CUDA library provides routines for finding out the number of GPUs in a system, and their properties.
- cudaGetDeviceCount(int *dev_count);
- cudaGetDeviceProperties(cudaDeviceProp *dev_prop, int n)
- cudaGetDeviceProperties populates a C struct with properties of the GPU.
- cudaSetDevice(n) selects GPU n for use.

query.cu

```
#include <cuda.h>
#include <stdio.h>
int main (int argc, char **argv){
    int ndev, maxtpb;
    cudaGetDeviceCount(&ndev);
    printf("Number of GPUs = %4d\n",ndev);
    for(int i=0;i<ndev;i++){
      cudaDeviceProp deviceProps;
      cudaGetDeviceProperties(&deviceProps, i);
       maxtpb = deviceProps.maxThreadsPerBlock;
       printf("GPU device %4d:\n\tName: %s:\n", i,deviceProps.name);
       printf("\tCompute capabilities: SM %d.%d\n",
            deviceProps.major, deviceProps.minor);
       printf("\tMaximum number of threads per block: %4d\n",maxtpb);
       printf("\tMaximum number of threads per SM: %4d\n",
            deviceProps.maxThreadsPerMultiProcessor);
       printf("\tNumber of streaming multiprocessors: %4d\n",
            deviceProps.multiProcessorCount);
       printf("\tClock rate: %d KHz\n",deviceProps.clockRate);
       printf("\tGlobal memory: %lu bytes\n",
            deviceProps.totalGlobalMem);
    }
    cudaSetDevice(0);
}
```

Output on labx03

```
Number of GPUs =    1
GPU device    0:
        Name: GeForce GTX 2700:
        Compute capabilities: SM 7.5
        Maximum number of threads per block:  1024
        Maximum number of threads per SM:  1024
        Number of streaming multiprocessors:    36
        Clock rate: 1620000 KHz
        Global memory: 8335327232 bytes
```

Can also get GPU information using nvidia-smi at command line