ENGN 2912B – Parallel and Hybrid Programming

(Slides borrowed from John Urbanic, Mauricio Ponga and Alex Razoumov)

December 5, 2018

Outline :

- Introduction to Hybrid Programming
  - OpenMP + MPI
  - OpenMP/MPI + CUDA
- Parallel Computing with OpenACC
  - Compiler Directives
  - Data Dependencies
  - 2-D Heat Flow Example (Laplace Equation)

Dhananjay Bhaskar

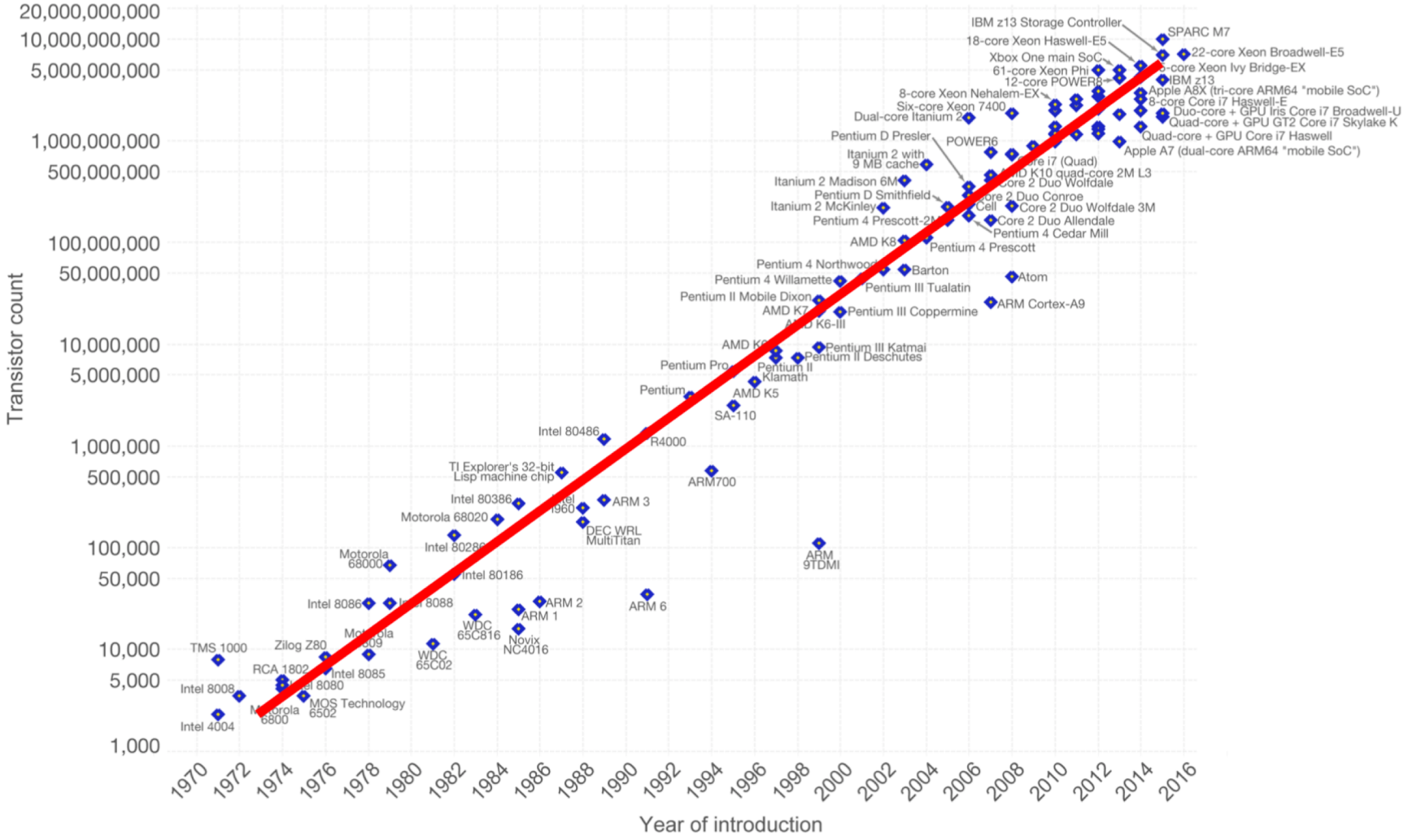
Ph.D. Student (2nd year), Biomedical Engineering

M.Sc., Applied Mathematics, UBC

B.Sc., Computer Science & Mathematics, UBC

BROWN

Moore's Law



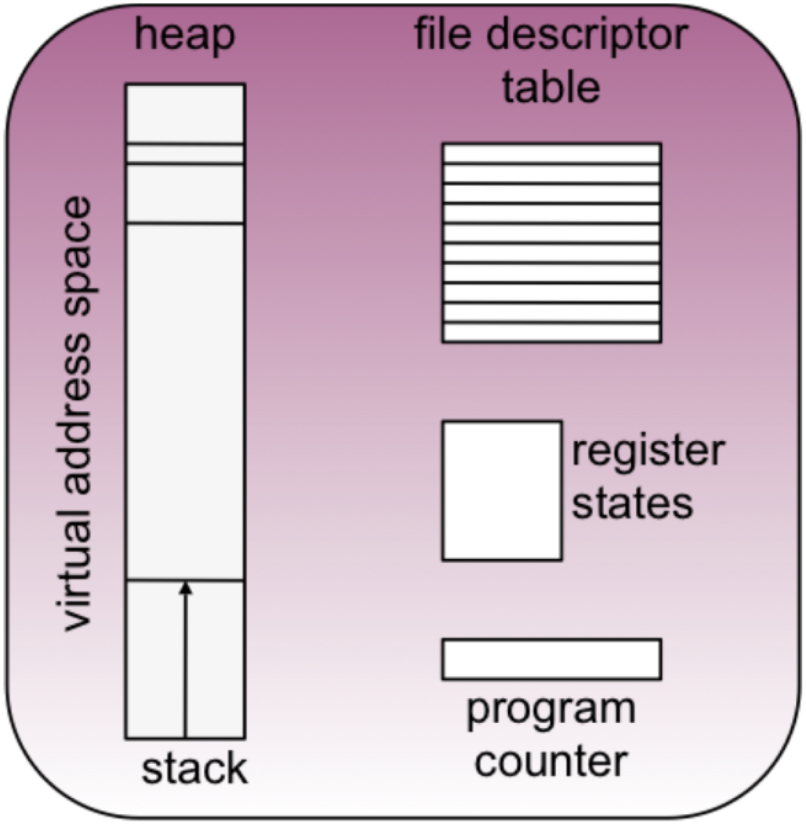
- multi-core machines are now relatively common in the consumer market and are quickly becoming commodity computing resources on large clusters
- all cores in a processor have access to shared memory

Parallel Programming Paradigms

- How do you take advantage of multiple CPUs/cores in a single system image?
  - multi-programming: start multiple processes to perform work simultaneously
  - multi-threading: introduce multiple threads of control within a single process to perform work simultaneously, threads will share memory and some other resources within the process

Processes

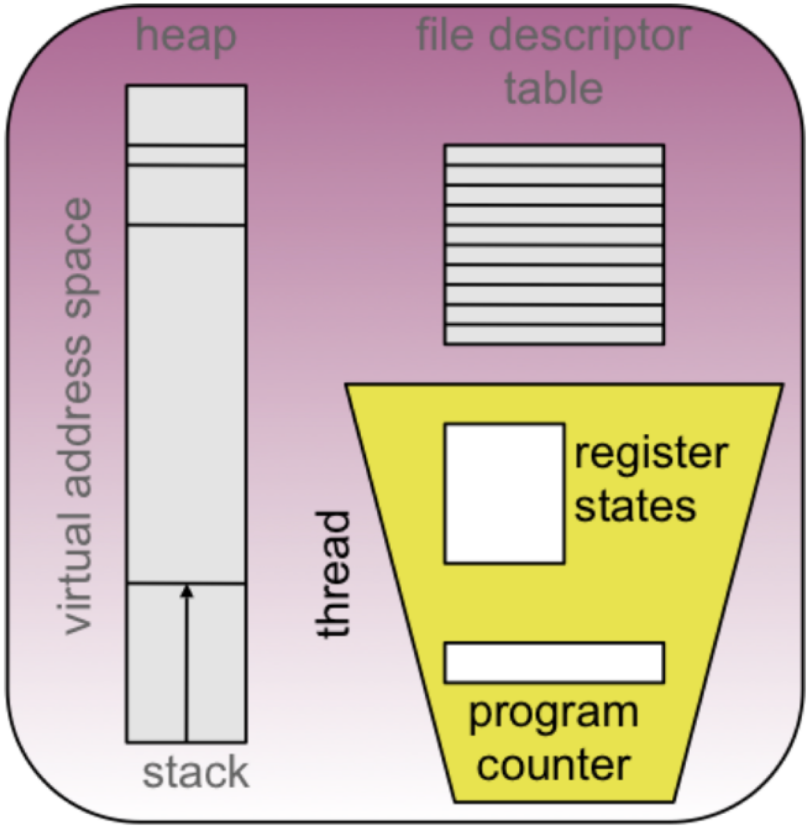
- DEF'N: a process is a program in execution, as seen by the OS
- A process imposes a specific set of management responsibilities on the OS
  - a unique virtual address space (**stack** for automatic variables of each function and for function arguments; **heap** for dynamic memory allocation; **code** itself)
  - file descriptor table
  - program counter (indicating where we are in the program)
  - register states (to save process when not running) – for task switching
  - shared libraries, etc.



Conceptual View of a Process (simplified)

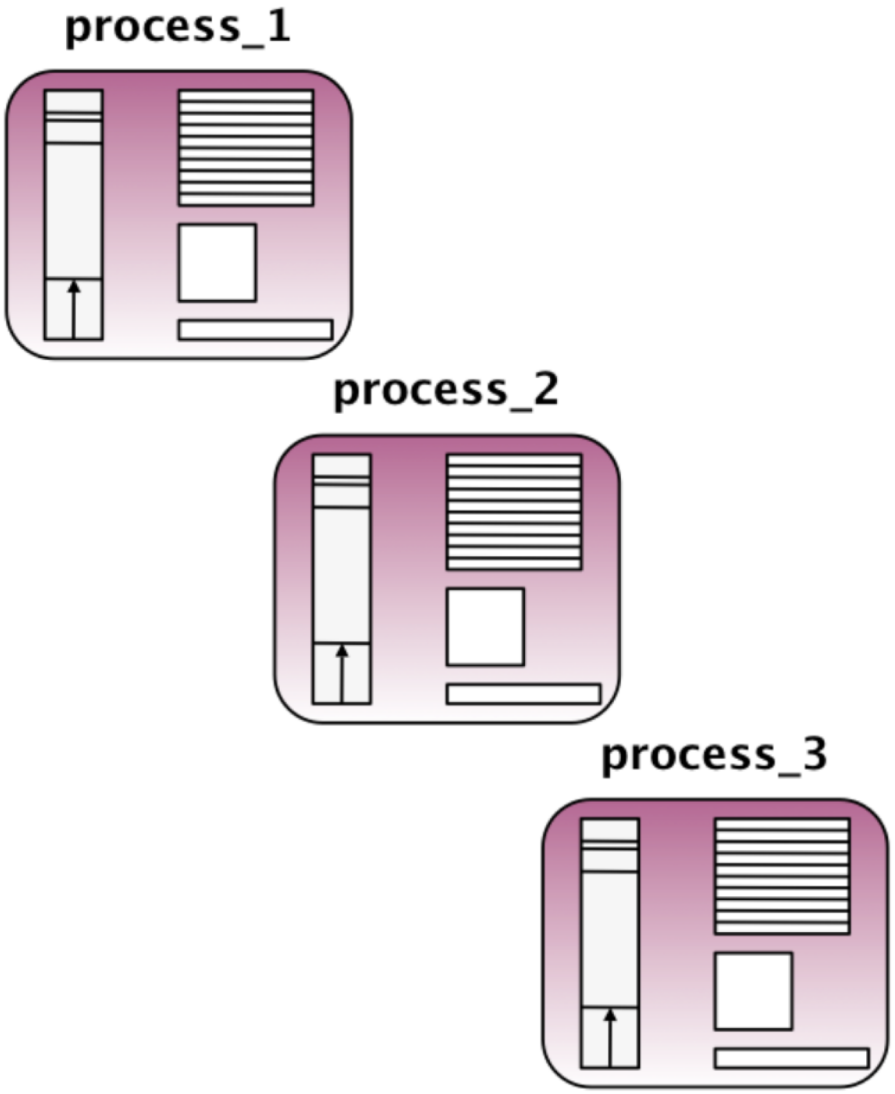
Threads

- DEF'N: a thread is a sequence of executable code within a process
- The smallest unit of processing that can be scheduled by the OS
- A serial process can be seen, at its simplest, as a single thread ("thread of control")
  - represented by the program counter (incremental for a sequence of instructions, iteration / conditional branch)
- In terms of record-keeping, only a small subset of a process is relevant when considering a thread
  (1) register states
  (2) program counter



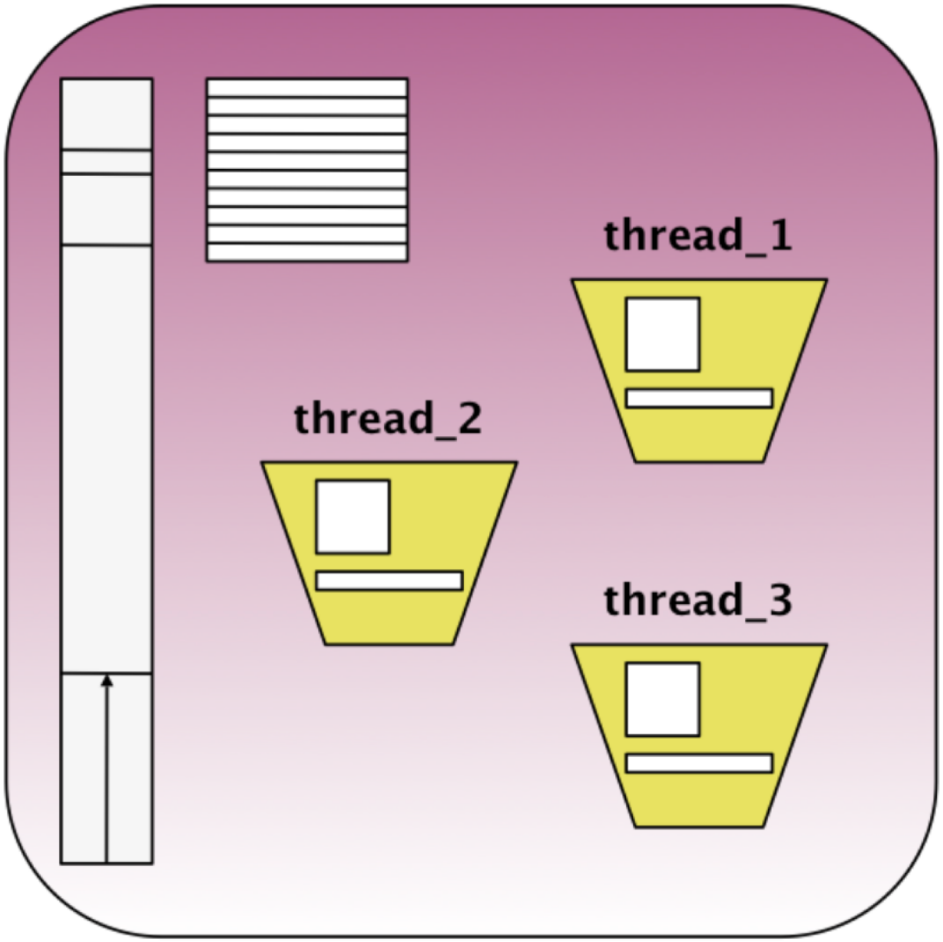
Conceptual View of a Thread (simplified)

Multi-Programming



- Distribute work by spawning multiple processes
  - e.g. fork(2), exec(2)
- Advantages
  - conceptually simple: each process is completely independent of others
  - process functionality can be highly cohesive
  - easily distributed
- Disadvantages
  - high resource cost
  - sharing file descriptors requires care and effort

Multi-Threading



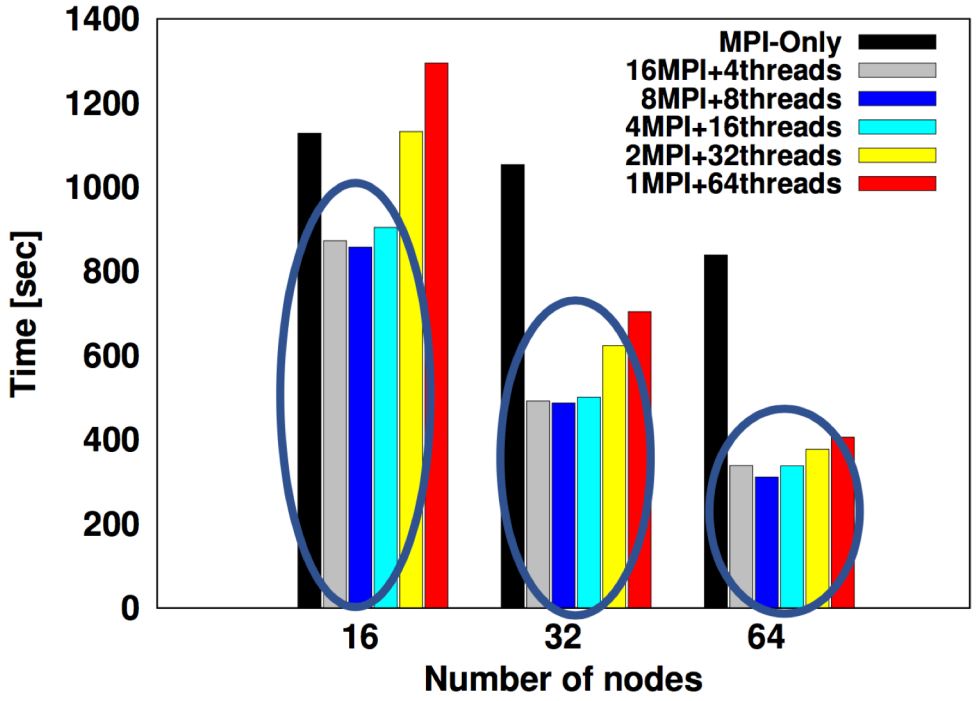
- Distribute work by defining multiple threads to do the work
  - e.g. pthreads
- Advantages
  - all process resources are implicitly shared (memory, file descriptors, etc.)
  - overhead incurred to manage multiple threads is relatively low
  - looks much like serial code
- Disadvantages
  - all data being implicitly shared creates a world of hammers, and your code/data is the thumb
  - exclusive access, contention, etc.

Hybrid Programming

- Combination of several *parallel* programming models in the same program
  - May be mixed in the same source
  - May be combinations of components or routines, each of which is in a single parallel programming model

---

- MPI + Threads or MPI + OpenMP is the most familiar hybrid model



IBM Blue-Gene/Q
16 cores per node

Optimal: 8x8

Hybrid Programming (MPI + OpenMP/Pthreads)

- MPI describes parallelism between processes (with separate address spaces)
- Thread parallelism provides a shared-memory model within a process
- OpenMP and Pthreads are common models
  - OpenMP provides convenient features for loop-level parallelism. Threads are created and managed by the compiler, based on user directives.
  - Pthreads provide more complex and dynamic approaches. Threads are created and managed explicitly by the user.

In Practice:

- Each MPI process is responsible for computing a small subset of data
- Each MPI process spawns a master thread
- The master thread manages worker threads by dynamically assigning tasks

Hybrid Programming (OpenMP + CUDA)

```
int main(int argc, char** argv) {

    int N = 2;
    size_t size = MSIZE * MSIZE * sizeof(float);
    int dev_count = 0;
    cudaGetDeviceCount(&dev_count);

    float * matAs[N], * matAs_h[N];
    float * matBs[N], * matBs_h[N];
    float * matCs[N], * matCs_h[N];
    srand(time(NULL));

    #pragma omp parallel for num_threads(5)
    for (int d=0; d<N; d++)
    {

        cudaSetDevice(d % dev_count);

        // Allocate space for the matrices
        cudaMalloc(&matAs[d], size);
        cudaMalloc(&matBs[d], size);
        cudaMalloc(&matCs[d], size);

        matAs_h[d] = (float *)malloc(size);
        matBs_h[d] = (float *)malloc(size);
        matCs_h[d] = (float *)malloc(size);

        // Seed the random number generator
        RandomFillKernel<<<MSIZE*MSIZE/BLOCK_SIZE,BLOCK_SIZE>>>(matAs[d],rand());
        RandomFillKernel<<<MSIZE*MSIZE/BLOCK_SIZE,BLOCK_SIZE>>>(matBs[d],rand());

        //Multiply the matrices
        MatMul(matAs[d], matBs[d], matCs[d]);

        //Copy the results back to host
        cudaMemcpy(matAs[d],matAs_h[d],size,cudaMemcpyDeviceToHost);
        cudaMemcpy(matBs[d],matBs_h[d],size,cudaMemcpyDeviceToHost);
        cudaMemcpy(matCs[d],matCs_h[d],size,cudaMemcpyDeviceToHost);

        cudaFree(matCs[d]);
        cudaFree(matBs[d]);
        cudaFree(matAs[d]);

        free(matAs_h[d]);
        free(matBs_h[d]);
        free(matCs_h[d]);
    }

    return 0;
}
```

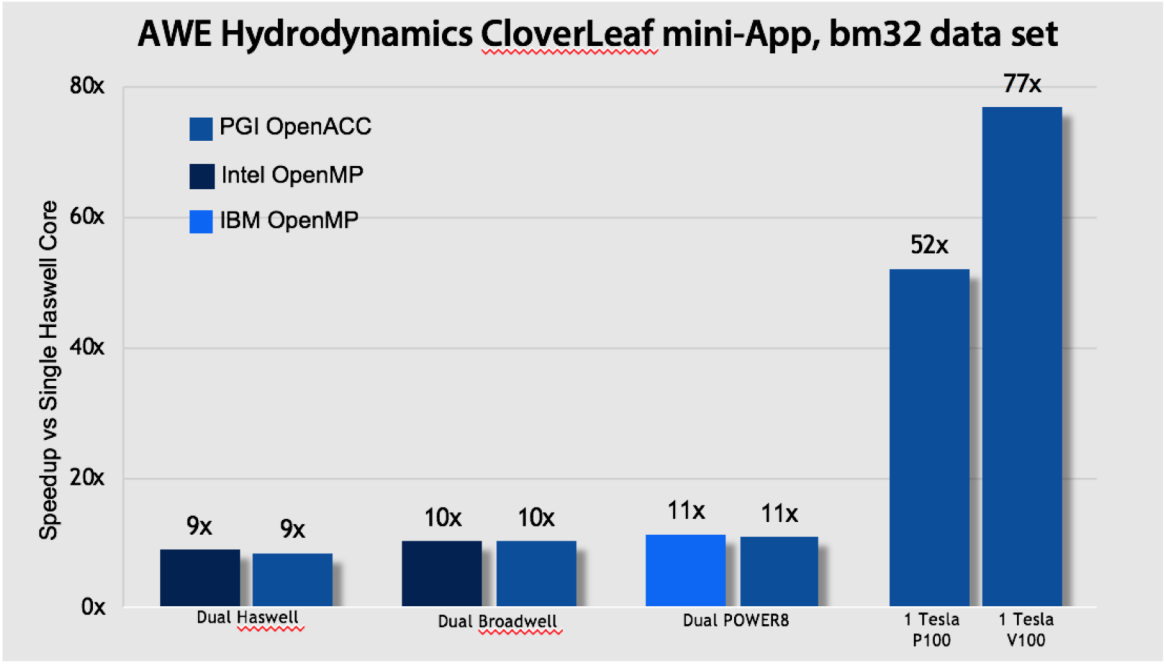
```
// Matrix multiplication kernel called by MatMul()
__global__ void MatMulKernel(float* A, float* B, float* C) {
    // Each thread computes one element of C by accumulating results into Cvalue
    float Cvalue = 0;

    // Compute the thread index
    int col = threadIdx.x + blockIdx.x * blockDim.x;
    int row = threadIdx.y + blockIdx.y * blockDim.y;
    // Compute the row and column
    for (int i = 0; i < MSIZE; ++i) {
        Cvalue += A[row * MSIZE + i] * B[i * MSIZE + col];
    }
    C[row*MSIZE+col] = Cvalue;
}

/**
 * Matrix multiplication.
 * Matrix dimensions are assumed to be multiples of BLOCK_SIZE
 */
void MatMul(float* A, float* B, float* C) {
    // Invoke kernel
    dim3 dimBlock(BLOCK_SIZE,BLOCK_SIZE);
    dim3 dimGrid(MSIZE/BLOCK_SIZE,MSIZE/BLOCK_SIZE);
    MatMulKernel<<<dimGrid,dimBlock>>>(A, B, C);
    cudaDeviceSynchronize();
}
```

Introduction to OpenACC

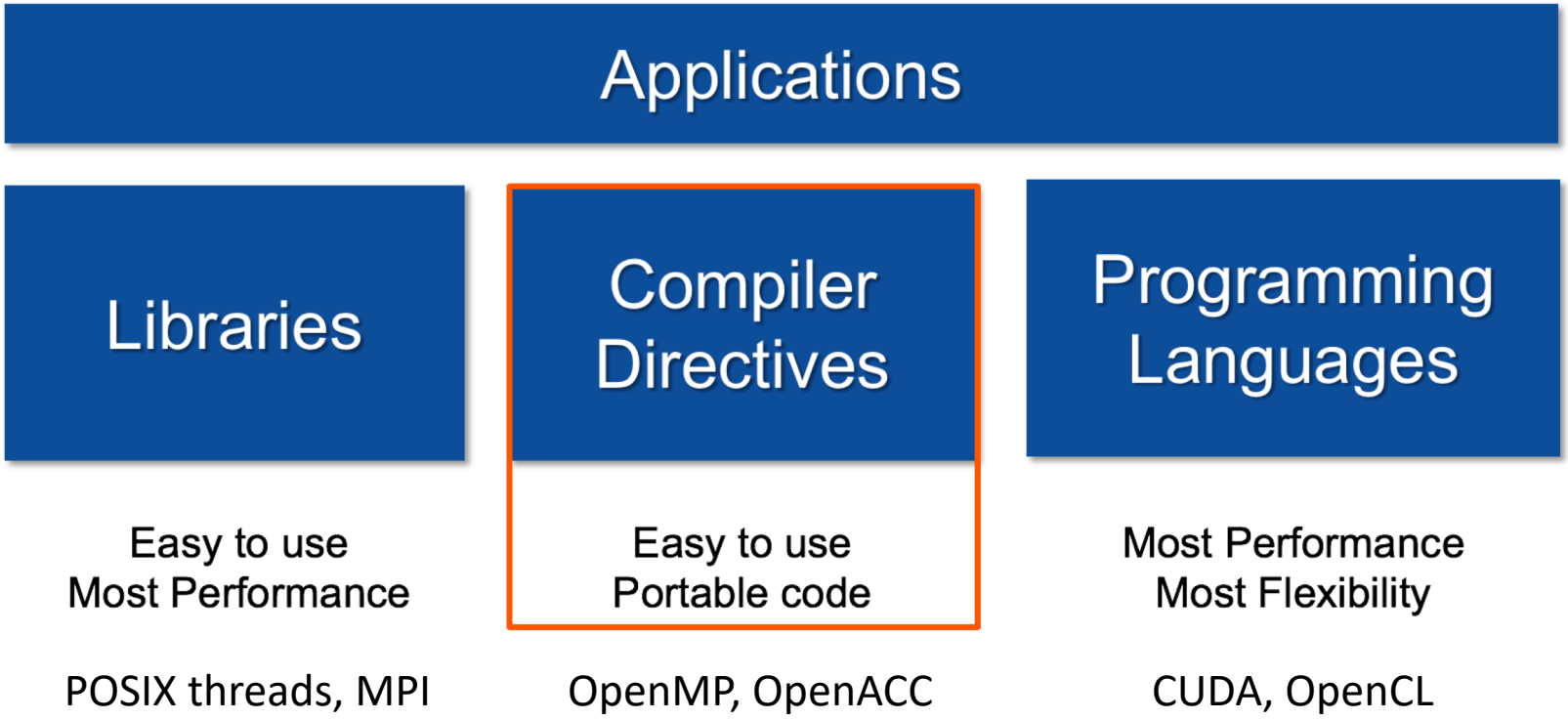
OpenACC is a directives-based programming approach to parallel computing designed for performance and portability on CPUs and GPUs.



Add Simple Compiler Directive

```
main()
 {
   <serial code>
   #pragma acc kernels
   {
     <parallel code>
   }
}
```

Accelerating Applications



Advantages of OpenACC

- **High-level**. Minimal modifications to the code. Less than with OpenCL, CUDA, etc. Easy to use for non-GPU programmers.
- **Single source**. No GPU-specific code. Compile the same program for accelerators or serial computation.
- **Efficient**. Experience shows very favorable comparison to low-level implementations of same algorithms.
- **Performance portable**. Supports CPUs, GPU accelerators and co-processors from multiple vendors, current and future versions.

A Simple Application

```
#include <stdlib.h>
#include <openacc.h>

void foo(int n, double a, double * x, double *restrict y)
{
  #pragma acc kernels
  for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}

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

  int N = 1 << 20;

  if (argc > 1)
    N = atoi(argv[1]);

  double *x = new double[N];
  double *y = new double[N];

  for (int i = 0; i < N; ++i) {
    x[i] = 2.0;
    y[i] = 1.0;
  }

  foo(N, 3.0, x, y);

  return 0;

}
```

module load pgi/16.7
export MP_BIND=no

pgc++ -acc -Minfo=accel -ta=multicore helloworld.cpp -o helloworld

pgc++ -acc -Minfo=accel -ta=tesla helloworld.cpp -o helloworld

-ta=tesla will *only* target a GPU
-ta=multicore will only target a multicore CPU
-Minfo=accel turns on helpful compiler reporting

Data Dependencies

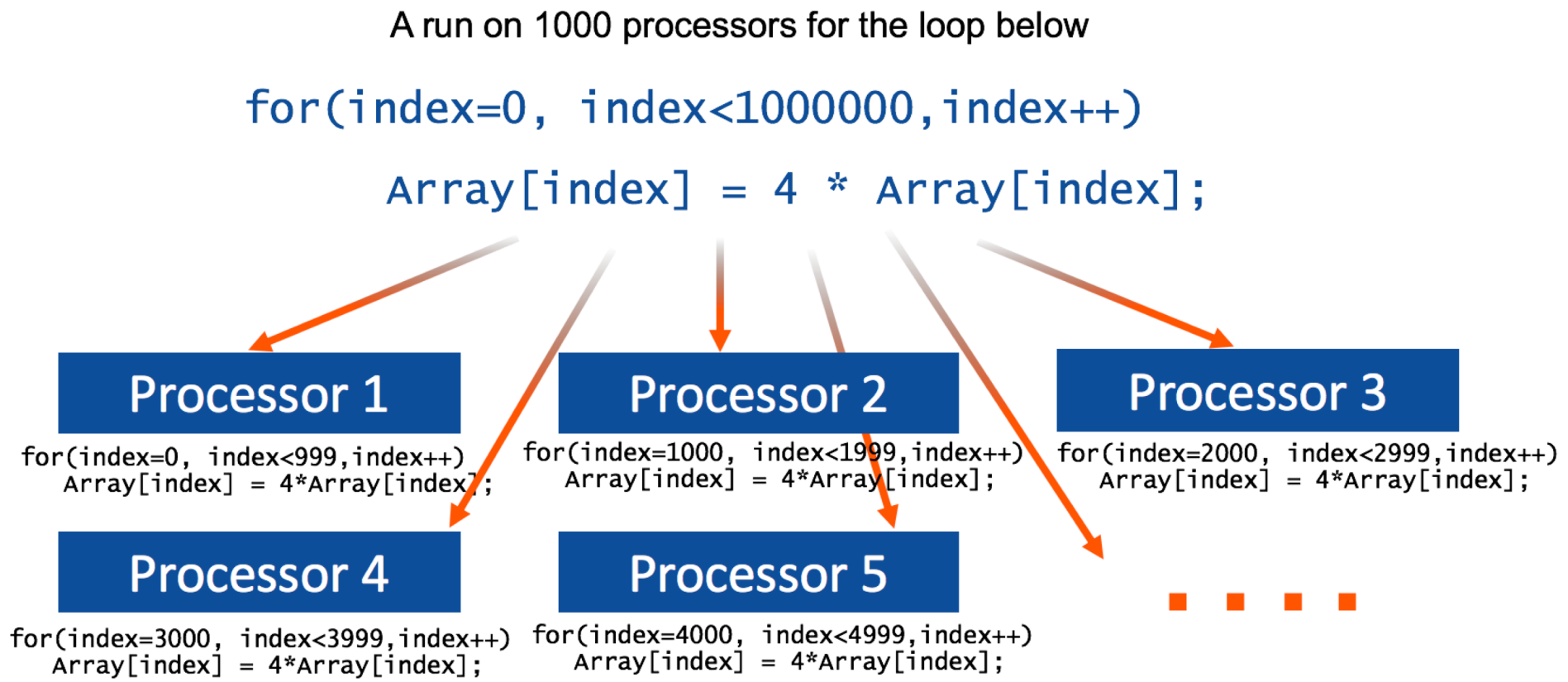
Most directive based parallelization consists of splitting up big do/for loops into independent chunks that the many processors can work on simultaneously.

Take, for example, a simple for loop like this:

```
for(index=0, index<1000000,index++)
        Array[index] = 4 * Array[index];
```

When run on 1000 processors, it will execute something like this…

Data Dependencies



Data Dependencies

But what if the loops are not entirely independent?

Take, for example, a similar loop like this:

```
for(index=1, index<1000000,index++)
      Array[index] = 4 * Array[index] – Array[index-1];
```

This is a perfectly valid serial code.

Data Dependencies

Processor 1

```
for(index=0, index<999,index++)
   Array[index] = 4*Array[index]-
Array[index-1];
```

Processor 2

```
for(index=1000, index<1999,index++)
   Array[index] = 4*Array[index]-
Array[index-1];
```

Result from Processor 1

```
for(index=1000, index<1999,index++)
      Array[1000] = 4 * Array[1000] – Array[999];
```

Needs the result of Processor 1's last iteration.

If we want the correct ("same as serial") result, we need to wait until processor 1 finishes. Likewise for processors 3, 4, …

Managing Data Dependencies

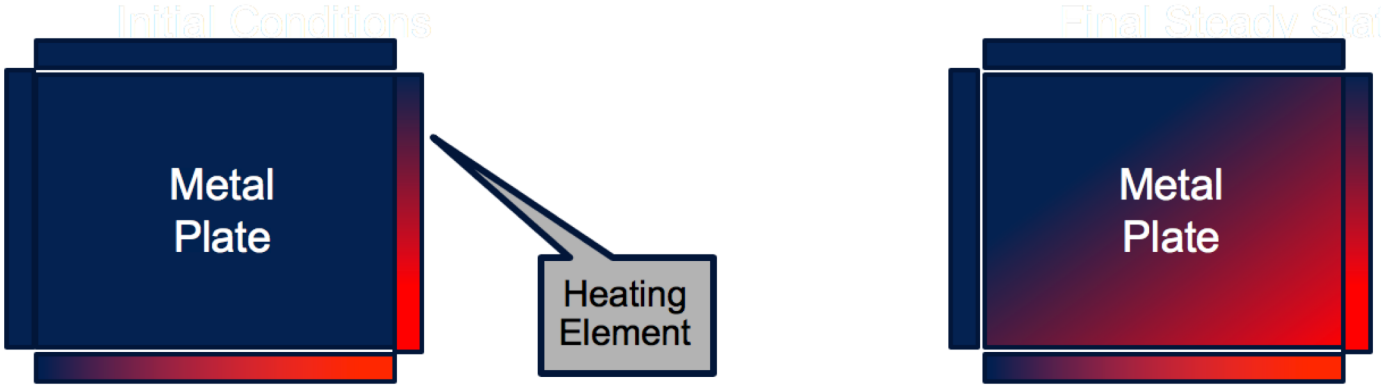
If the compiler even *suspects* that there is a data dependency, it will, for the sake of correctness, refuse to parallelize that loop.

```
11, Loop carried dependence of 'Array' prevents parallelization
    Loop carried backward dependence of 'Array' prevents vectorization
```

- Rearrange your code to make it more obvious to the compiler that there is not really a data dependency.
- Eliminate a real dependency by changing your code.
- There is a common bag of tricks developed for this as this issue goes back 40 years in HPC. Many are quite trivial to apply.
- The compilers have gradually been learning these themselves.
- Override the compiler's judgment (`independent` clause) at the risk of invalid results. Misuse of `restrict` has similar consequences.

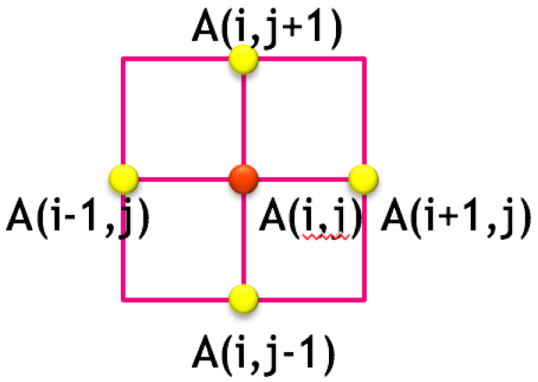
Example: Heat Flow

- Solve the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including: electrostatics, fluid flow, and temperature
- For temperature, it is the Steady State Heat Equation:



Numerical Solution for Steady State

- The Laplace equation on a grid states that each grid point is the average of it's neighbors.
- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.
- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate.



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

C++

```
for(i = 1; i <= ROWS; i++) {
     for(j = 1; j <= COLUMNS; j++) {
          Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                      Temperature_last[i][j+1] + Temperature_last[i][j-1]);
     }
}
```

Initialization and Boundary Conditions

```
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

Serial Solution

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

     for(i = 1; i <= ROWS; i++) {
          for(j = 1; j <= COLUMNS; j++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

     for(i = 1; i <= ROWS; i++){
          for(j = 1; j <= COLUMNS; j++){
         dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
         Temperature_last[i][j] = Temperature[i][j];
          }
     }

     iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Parallel Solution

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    #pragma acc kernels
     for(i = 1; i <= ROWS; i++) {
         for(j = 1; j <= COLUMNS; j++) {
             Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                         Temperature_last[i][j+1] + Temperature_last[i][j-1]);
         }
     }

     dt = 0.0; // reset largest temperature change

    #pragma acc kernels
    {
      for(i = 1; i <= ROWS; i++){
         for(j = 1; j <= COLUMNS; j++){
             dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
             Temperature_last[i][j] = Temperature[i][j];
         }
     }
    }
     if((iteration % 100) == 0) {
         track_progress(iteration);
     }

     iteration++;
}
```

Generate a parallel Kernel

Generate a parallel Kernel

CPU PERFORMANCE

3372 steps to convergence

Execution	Problem Size			
	1000x1000 (Lab1)		2000x2000	
	Time (s)	Speedup	Time (s)	Speedup
CPU Serial	6.2	--	40.3	--
OpenACC 2 CPU Cores	3.0	2.07	26.2	1.54
OpenACC 4 CPU Cores	1.5	4.13	21.3	1.89

Running on GPU

```
% ./laplace.out
…
---------- Iteration number: 3100 ------------
[995,995]: 97.58  [996,996]: 98.18  [997,997]: 98.71  [998,998]: 99.17  [999,999]: 99.55
[1000,1000]: 99.86
---------- Iteration number: 3200 ------------
[995,995]: 97.62  [996,996]: 98.21  [997,997]: 98.73  [998,998]: 99.18  [999,999]: 99.56
[1000,1000]: 99.86
---------- Iteration number: 3300 ------------
[995,995]: 97.66  [996,996]: 98.24  [997,997]: 98.75  [998,998]: 99.19  [999,999]: 99.56
[1000,1000]: 99.87

Max error at iteration 3372 was 0.009995
Total time was 40.462415 seconds.
```

Profiling GPU Performance

```
% pgprof –cpu-profiling-mode top-down ./laplace.out
….

==455== Profiling result:
Time(%)     Time     Calls       Avg       Min       Max  Name
 57.41%  13.2519s     13488  982.49us     959ns  1.3721ms  [CUDA memcpy HtoD]
 36.08%  8.32932s     10116  823.38us  2.6230us  1.5129ms  [CUDA memcpy DtoH]
  3.42%  788.42ms      3372  233.81us  231.79us  235.79us  main_80_gpu
  2.80%  646.58ms      3372  191.75us  189.69us  194.04us  main_69_gpu
  0.29%  66.916ms      3372  19.844us  19.519us  20.223us  main_81_gpu_red

======== CPU profiling result (top down):
Time(%)     Time  Name
…
  9.15%  16.3609s  | __pgi_uacc_dataexitdone
  9.11%  16.2809s  | __pgi_uacc_dataonb
…
```

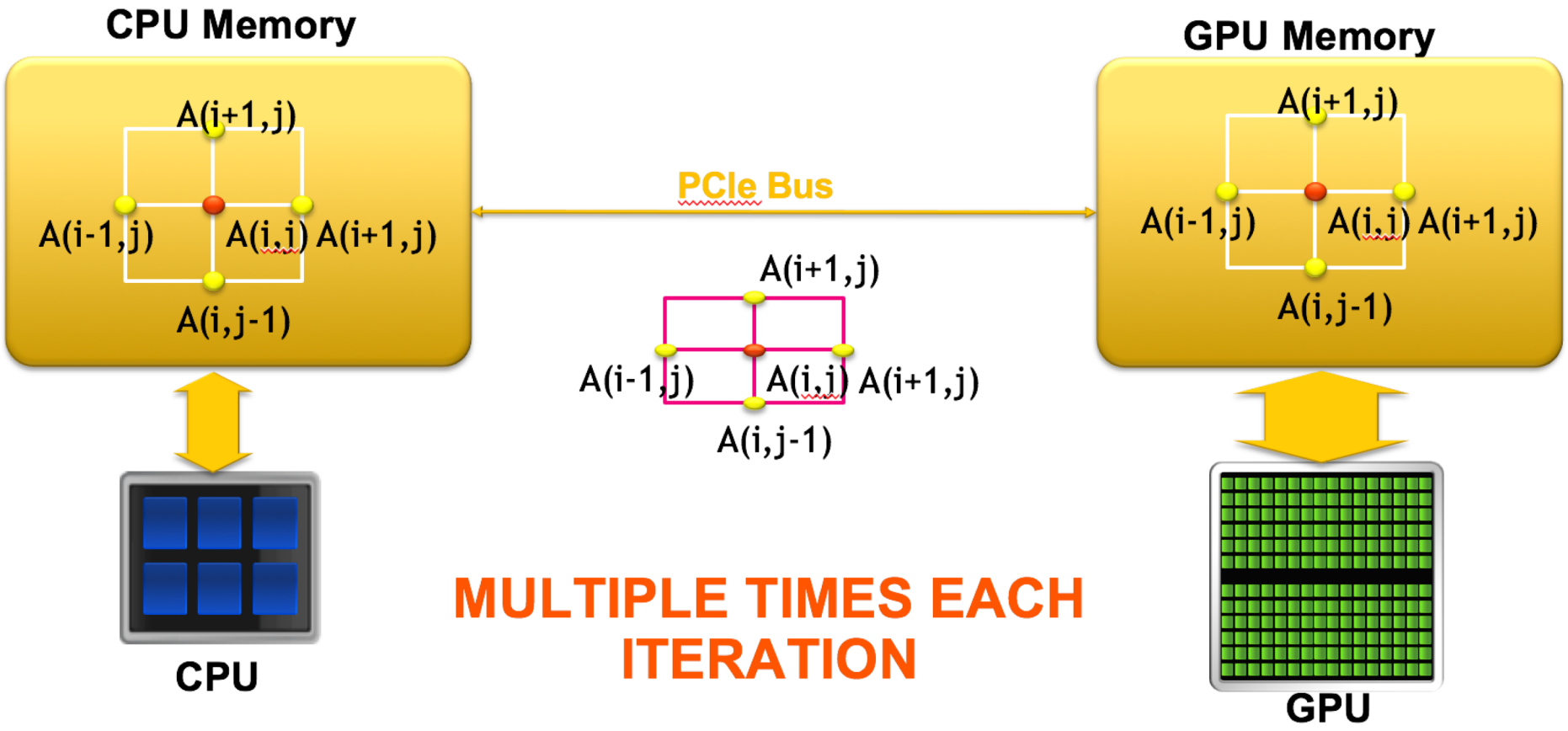
13.2 seconds copying data from host to device

8.3 seconds copying data from device to host

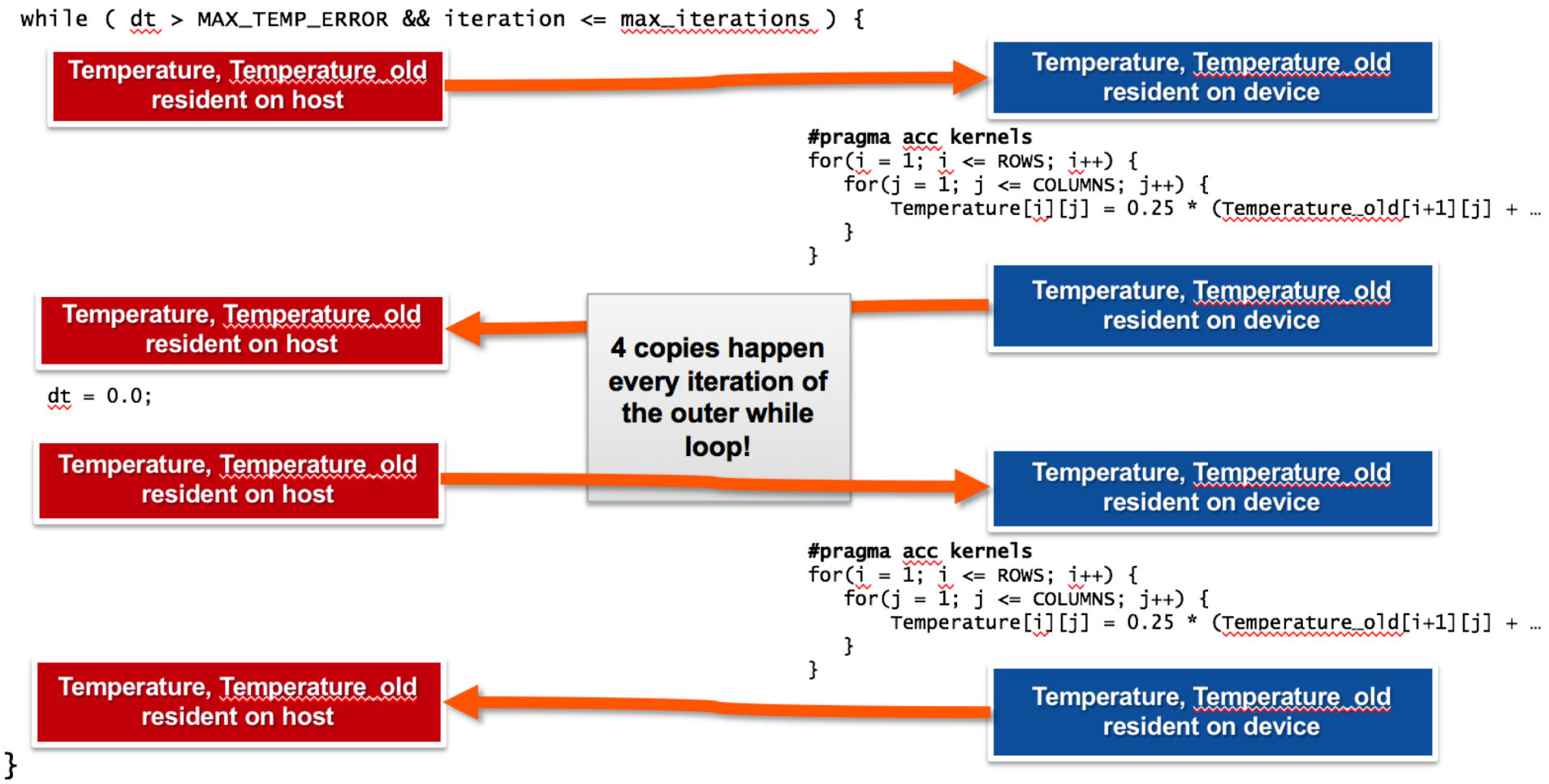
Including the copy, there is an additional CPU overhead needed to create and delete the device data

OpenACC
More Science, Less Programming

Inefficient Data Transfer



Inefficient Data Transfer



Data Management & Data Clauses

```
#pragma acc data [clause …]
{

    structured block

}
```

copy(*list*) — **Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.**

Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.

copyin(*list*) — **Allocates memory on GPU and copies data from host to GPU when entering region.**

Principal use: Think of this like an array that you would use as just an input to a subroutine.

copyout(*list*) — **Allocates memory on GPU and copies data to the host when exiting region.**

Principal use: A result that isn't overwriting the input data structure.

create(*list*) — **Allocates memory on GPU but does not copy.**

Principal use: Temporary arrays.

Specifying Data Regions

- Compilers sometimes cannot determine the size of arrays, so we must specify explicitly using data clauses with an array “shape”. The compiler will let you know if you need to do this. Sometimes, you will want to for your own efficiency reasons.

C++ (start:count)

```
#pragma acc data copyin(a[0:count]), copyout(b[s/4:3*s/4])
```

C++ Code

```
int main(int argc, char *argv[]) {

  int i;
  double A[2000], B[1000], C[1000];

  #pragma acc kernels
  for (i=0; i<1000; i++){

    A[i] = 4 * i;
    B[i] = B[i] + 2;
    C[i] = A[i] + 2 * B[i];

  }
}
```

Compiler Output

```
pgcc -acc -Minfo=accel loops.c
main:
      6, Generating implicit copy(B[:])
         Generating implicit copyout(C[:])
         Generating implicit copyout(A[:1000])
      7, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
          7, #pragma acc loop gang, vector(128)
```

Specifying Data Regions

Simplest Kernel

```
int main(int argc, char** argv){

float A[1000];


    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
      A[iter] = 1.0;
    }

    A[10] = 2.0;


  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

A[ ] Copied To Host

Runs On Host

Output:
A[10] = 2.0

With Global Data Region

```
int main(int argc, char** argv){

float A[1000];

#pragma acc data copy(A)
{

    #pragma acc kernels
    for( int iter = 1; iter < 1000 ; iter++){
      A[iter] = 1.0;
    }

    A[10] = 2.0;

}

  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

Still Runs On Host

A[ ] Copied To Host

Output:
A[10] = 1.0

GPU Solution

```
#pragma acc data create(Temperature_last[0:ROWS+2][0:COLUMNS+2], Temperature[0:ROWS+2][0:COLUMNS+2])
{
    initialize();                   // initialize Temp_last including boundary conditions

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        #pragma acc kernels present(Temperature_last, Temperature)
        {
        for(i = 1; i <= ROWS; i++) { for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }}

        dt = 0.0; // reset largest temperature change

        // copy grid to old grid for next iteration and find latest dt
        for(i = 1; i <= ROWS; i++){ for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }}
        } // end acc kernels
        // periodically print test values
        if((iteration % 100) == 0) {
#pragma acc update self(Temperature[0:ROWS+2][0:COLUMNS+2])
          track_progress(iteration);
        }

      iteration++;
    }
} // end acc data region
```

Allocate space on the device

Generate parallel kernels

Periodically synchronize data between host and device

Performance Comparison

Run	1000x1000		2000x2000	
	Time (s)	Speed-up	Time (s)	Speed-up
Serial CPU	6.2	1.0X	40.3	1.0X
OpenACC 4-core CPU	1.5	4.1X	21	1.9X
OpenACC Tesla	1.6	3.9X	6.6	6.1X

Resources

- Asynchronous OpenACC:

https://developer.nvidia.com/openacc-overview-course Class #4

- Using OpenACC with MPI:

https://developer.nvidia.com/openacc-advanced-course Class #2

- Multi-GPU OpenACC:

http://on-demand-gtc.gputechconf.com/gtc-quicklink/hhZdn

Resources

https://www.openacc.org/resources