

# Introduction to OpenACC

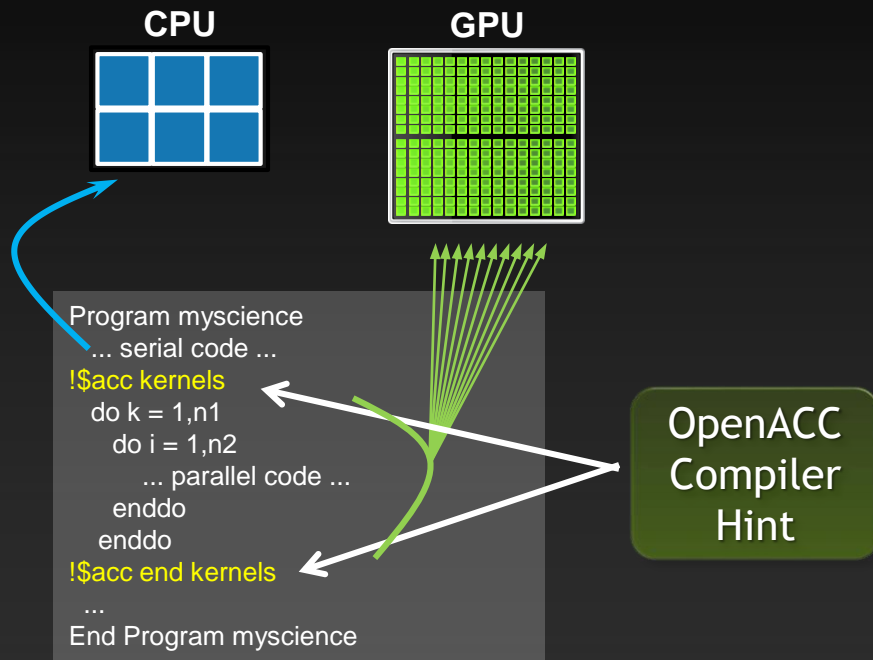
John Urbanic

Parallel Computing Scientist  
Pittsburgh Supercomputing Center

# What is OpenACC?

*It is a directive based standard to allow developers to take advantage of accelerators such as GPUs from NVIDIA and AMD, Intel's Xeon Phi, FPGAs, and even DSP chips.*

# Directives



Simple compiler hints from coder.

Compiler generates parallel threaded code.

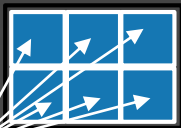
Ignorant compiler just sees some comments.

Your original  
Fortran or C code

# Familiar to OpenMP Programmers

## OpenMP

### CPU



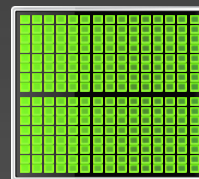
```
main() {  
    double pi = 0.0; long i;  
  
    #pragma omp parallel for reduction(+:pi)  
    for (i=0; i<N; i++)  
    {  
        double t = (double)((i+0.05)/N);  
        pi += 4.0/(1.0+t*t);  
    }  
  
    printf("pi = %f\n", pi/N);  
}
```

## OpenACC

### CPU



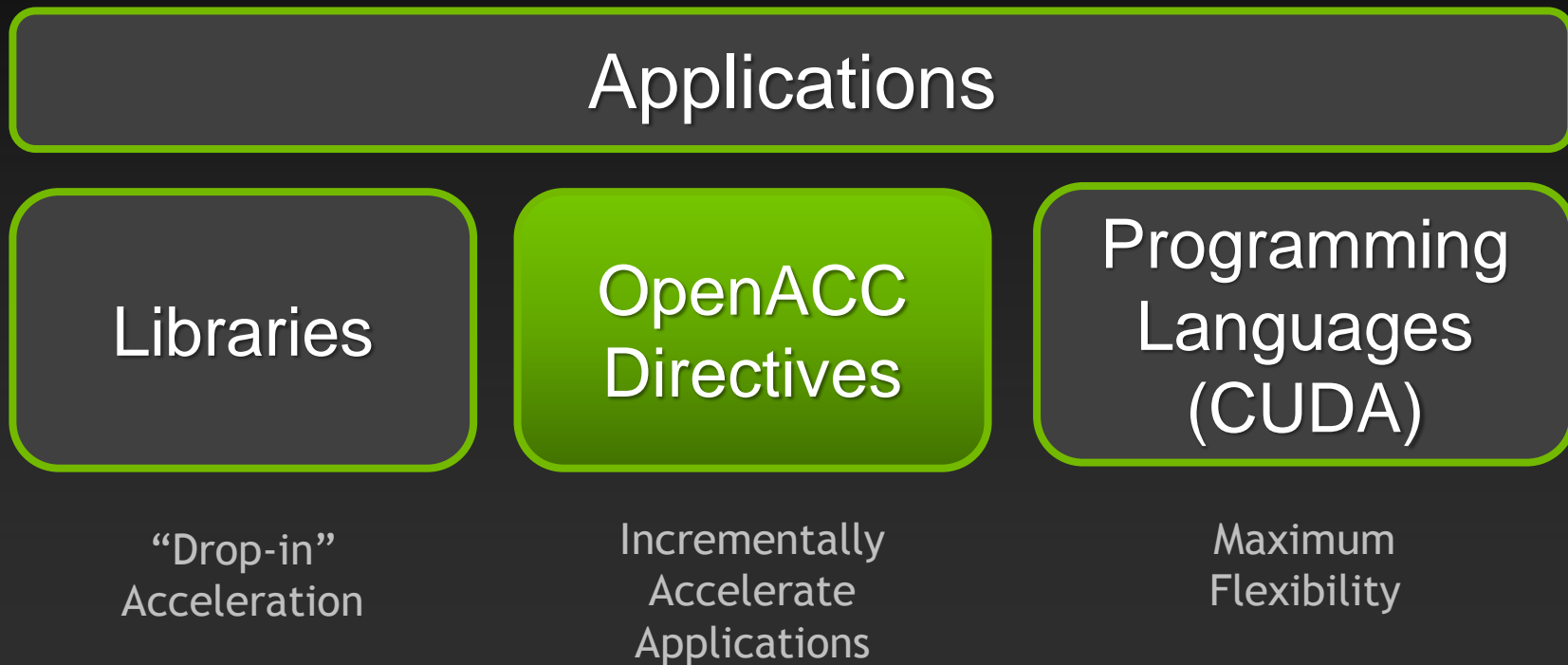
### GPU



```
main() {  
    double pi = 0.0; long i;  
  
    #pragma acc kernels  
    for (i=0; i<N; i++)  
    {  
        double t = (double)((i+0.05)/N);  
        pi += 4.0/(1.0+t*t);  
    }  
  
    printf("pi = %f\n", pi/N);  
}
```

More on this later!

# How Else Would We Accelerate Applications?



# Key Advantages Of This Approach

- High-level. No involvement of OpenCL, CUDA, etc.
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial, non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be quick.

# A Few Cases

Reading DNA nucleotide sequences

*Shanghai JiaoTong University*



**4 directives**

**16x faster**

Designing circuits for quantum computing

*UIST, Macedonia*



**1 week**

**40x faster**

Extracting image features in real-time

*Aselsan*



**3 directives**

**4.1x faster**

HydroC- Galaxy Formation

*PRACE Benchmark Code, CAPS*



**1 week**

**3x faster**

Real-time Derivative Valuation

*Opel Blue, Ltd*

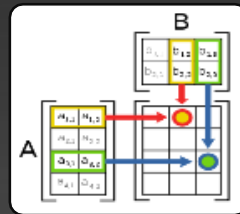


**Few hours**

**70x faster**

Matrix Matrix Multiply

*Independent Research Scientist*



**4 directives**

**6.4x faster**

# A Champion Case

**4x Faster**

**Jaguar**

42 days

**Titan**

10 days

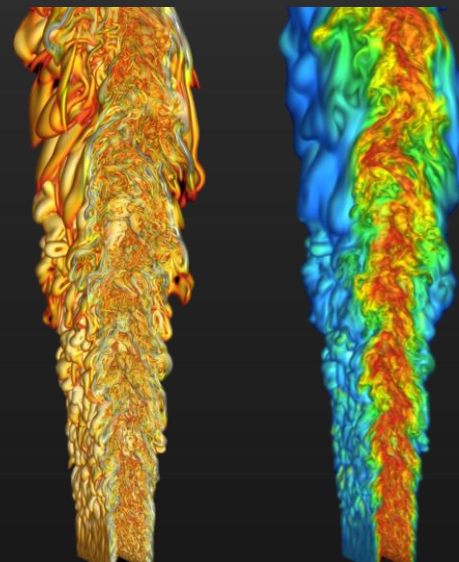
---

Modified <1%  
Lines of Code

---

15 PF! One of fastest  
simulations ever!

Design alternative fuels with  
up to 50% higher efficiency



**S3D: Fuel Combustion**



# Broad Accelerator Support

- Xeon Phi support already in CAPS. Demonstrated and soon to be release for PGI.
- AMD line of accelerated processing units (APUs) as well as the AMD line of discrete GPUs for preliminary PGI support.
- Carma - a hybrid platform based on ARM Cortex-A9 quad core and an NVIDIA Quadro® 1000M GPU.
- NVIDIA...

# NVIDIA Rules

or writes the rules. They have been the foremost supporter of GPU computing for much of the past decade, and have earned the focus of this workshop. We are using NVIDIA GPUs as our platform and our touchstone because:

- They are proven
- Well understood
- Best bang for buck if you want to buy an accelerator
- Excellent support by vendor and community
- It is the basis for our leading edge platform, Keeneland
- It will not be going obsolete any time soon
- NVIDIA recently acquired PGI. That gave us a slight preference for the PGI compiler over the Cray one. Both are available on Blue Waters.

# True Standard

- Full OpenACC 1.0 and 2.0 and now 2.5 Specifications available online

<http://www.openacc-standard.org>

- Quick reference card also available
- Implementations available now from PGI, Cray, and CAPS.
- GCC version of OpenACC now in 5.x

## The OpenACC™ API QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.



Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

# A Simple Example: SAXPY

## *SAXPY in C*

```
void saxpy(int n,  
          float a,  
          float *x,  
          float *restrict y)  
{  
    #pragma acc kernels  
    for (int i = 0; i < n; ++i)  
        y[i] = a*x[i] + y[i];  
}  
  
...  
// Somewhere in main  
// call SAXPY on 1M elements  
saxpy(1<<20, 2.0, x, y);  
...
```

## *SAXPY in Fortran*

```
subroutine saxpy(n, a, x, y)  
    real :: x(:), y(:), a  
    integer :: n, i  
    !$acc kernels  
    do i=1,n  
        y(i) = a*x(i)+y(i)  
    enddo  
    !$acc end kernels  
end subroutine saxpy  
  
...  
$ From main program  
$ call SAXPY on 1M elements  
call saxpy(2**20, 2.0, x_d, y_d)  
...
```

# kernel's: Our first OpenACC Directive

We request that each loop execute as a separate *kernel* on the GPU. This is an incredibly powerful directive.

```
!$acc kernels
```

```
  do i=1,n  
    a(i) = 0.0  
    b(i) = 1.0  
    c(i) = 2.0  
  end do
```



kernel 1

```
  do i=1,n  
    a(i) = b(i) + c(i)  
  end do
```



kernel 2

```
!$acc end kernels
```

## Kernel:

A parallel routine to run on the GPU

# General Directive Syntax and Scope

## Fortran

```
!$acc kernels [clause ...]  
    structured block  
!$acc end kernels
```

## C

```
#pragma acc kernels [clause ...]  
    {  
        structured block  
    }
```

I may indent the directives at the natural code indentation level for readability. It is a common practice to always start them in the first column (ala #define/#ifdef). Either is fine with C or Fortran 90 compilers.

# Complete SAXPY Example Code

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

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

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

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

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

```
#include <stdlib.h>

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

# Complete SAXPY Example Code

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

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

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

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

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

```
#include <stdlib.h>

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

"I promise y is not aliased by  
Anything else (esp. x)"



# C Detail: the restrict keyword

- Standard C (as of C99).
- Important for optimization of serial as well as OpenACC and OpenMP code.
- Promise given by the programmer to the compiler for a pointer

```
float *restrict ptr
```

Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”

- Limits the effects of pointer aliasing
- OpenACC compilers often require `restrict` to determine independence
  - Otherwise the compiler can't parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined

# Compile and Run

- C: `pgcc -acc -Minfo=accel saxpy.c`
- Fortran: `pgf90 -acc -Minfo=accel saxpy.f90`

## Compiler Output

```
pgcc -acc -Minfo=accel saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
    Generating copy(y[:n-1])
    Generating compute capability 1.0 binary
    Generating compute capability 2.0 binary
  9, Loop is parallelizable
    Accelerator kernel generated
    9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
      CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
      CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

- Run: `a.out`

# Compare: Partial CUDA C SAXPY Code

## Just the subroutine

```
__global__ void saxpy_kernel( float a, float* x, float* y, int n ){
    int i;
    i = blockIdx.x*blockDim.x + threadIdx.x;
    if( i <= n ) x[i] = a*x[i] + y[i];
}

void saxpy( float a, float* x, float* y, int n ){
    float *xd, *yd;
    cudaMalloc( (void**)&xd, n*sizeof(float) );
    cudaMalloc( (void**)&yd, n*sizeof(float) ); cudaMemcpy( xd, x, n*sizeof(float),
                                                            cudaMemcpyHostToDevice );
    cudaMemcpy( yd, y, n*sizeof(float),
               cudaMemcpyHostToDevice );
    saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
    cudaMemcpy( x, xd, n*sizeof(float),
               cudaMemcpyDeviceToHost );
    cudaFree( xd ); cudaFree( yd );
}
```

# Compare: Partial CUDA Fortran SAXPY Code

## Just the subroutine

```
module kmod
  use cudafor
contains
  attributes(global) subroutine saxpy_kernel(A,X,Y,N)
    real(4), device :: A, X(N), Y(N)
    integer, value :: N
    integer :: i
    i = (blockid%x-1)*blockdim%x + threadid%x
    if( i <= N ) X(i) = A*X(i) + Y(i)
  end subroutine
end module

subroutine saxpy( A, X, Y, N )
  use kmod
  real(4) :: A, X(N), Y(N)
  integer :: N
  real(4), device, allocatable, dimension(:):: &
    Xd, Yd
  allocate( Xd(N), Yd(N) )
  Xd = X(1:N)
  Yd = Y(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd, Yd, N)
  X(1:N) = Xd
  deallocate( Xd, Yd )
end subroutine
```

# Again: Complete SAXPY Example Code

## Main Code

```
int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats

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

    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));

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

    saxpy(N, 3.0f, x, y);

    return 0;
}
```

## Entire Subroutine

```
#include <stdlib.h>

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

# Big Difference!

- With CUDA, we changed the structure of the old code. Non-CUDA programmers can't understand new code. It is not even ANSI standard code.
- We have separate sections for the host code, and the GPU code. Different flow of code. Serial path now gone forever.
- Where did these “32's” and other mystery variables come from? This is a clue that we have some hardware details to deal with here.
- Exact same situation as assembly used to be. How much hand-assembled code is still being written in HPC now that compilers have gotten so efficient?

# This looks easy! Too easy...

- If it is this simple, why don't we just throw *kernel* in front of every loop?
- Better yet, why doesn't the compiler do this for me?

The answer is that there are two general issues that prevent the compiler from being able to just automatically parallelize every loop.

- Data Dependencies in Loops
- Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results, and reasonable performance.

# 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...



# No Data Dependency

Processor  
1

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

Processor  
2

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

Processor  
3

```
for(index=2000, index<2999,index++)  
  Array[index] = 4*Array[index];
```

Processor  
4

```
for(index=3000, index<3999,index++)  
  Array[index] = 4*Array[index];
```

Processor  
5

```
for(index=4000, index<4999,index++)  
  Array[index] = 4*Array[index];
```



# Data Dependency

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 perfectly valid serial code.

# Data Dependency

Now Processor 2, in trying to calculate its first iteration...

```
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, ...

# Data Dependencies

That is a data dependency. 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

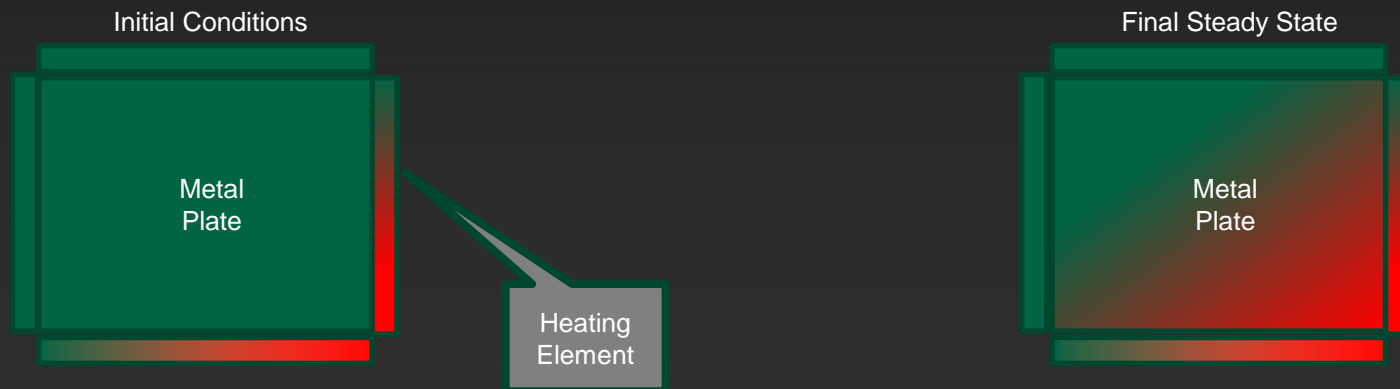
As large, complex loops are quite common in HPC, especially around the most important parts of your code, the compiler will often balk most when you most need a kernel to be generated. What can you do?

# Data Dependencies

- 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.

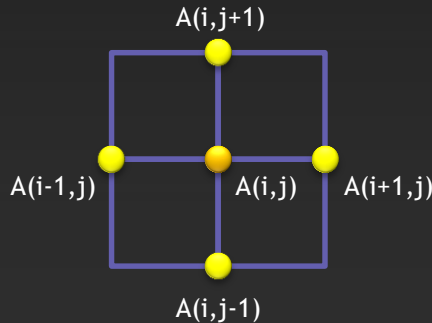
# Our Foundation Exercise: Laplace Solver

- I've been using this for MPI, OpenMP and now OpenACC. It is a great simulation problem, not rigged for OpenACC.
- In this most basic form, it solves the Laplace equation:  $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
  - Temperature
- For temperature, it is the Steady State Heat Equation:



# Exercise Foundation: Jacobi Iteration

- The Laplace equation on a grid states that each grid point is the average of its 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.



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

# Serial Code Implementation

```
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]);  
    }  
}
```

```
do j=1,columns  
    do i=1,rows  
        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) )  
    enddo  
enddo
```



# Serial C Code (kernel)

```
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];  
        }  
    }
```

```
    if((iteration % 100) == 0) {  
        track_progress(iteration);  
    }
```

```
    iteration++;
```

```
}
```



**Done?**



**Calculate**



**Update  
temp  
array and  
find max  
change**



**Output**

# Serial C Code Subroutines

```
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;
    }
}
```

```
void track_progress(int iteration) {
    int i;

    printf("-- Iteration: %d --\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

BCs could run from 0 to ROWS+1 or from 1 to ROWS. We chose the former.

## Whole C Code

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS 1000
#define ROWS 1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2]; // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

// helper routines
void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {
    int i, j; // grid indexes
    int max_iterations; // number of iterations
    int iteration=1; // current iteration
    double dt=100; // largest change in t
    struct timeval start_time, stop_time, elapsed_time; // timers

    printf("Maximum iterations [100-4000]? \n");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time, NULL); // Unix timer

    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
        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];
            }
        }

        // periodically print test values
        if((iteration % 100) == 0) {
            track_progress(iteration);
        }

        iteration++;
    }
}
```

```
gettimeofday(&stop_time, NULL);
timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine

printf("\nMax error at iteration %d was %f\n", iteration-1, dt);
printf("Total time was %f seconds.\n", elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);
}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
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;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    int i;

    printf("----- Iteration number: %d ----- \n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

# Serial Fortran Code (kernel)

```
do while ( dt > max_temp_error .and. iteration <= max_iterations)
```

```
  do j=1,columns
    do i=1,rows
      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) )
    enddo
  enddo
```

```
  dt=0.0
```

```
  do j=1,columns
    do i=1,rows
      dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
      temperature_last(i,j) = temperature(i,j)
    enddo
  enddo
```

```
  if( mod(iteration,100).eq.0 ) then
    call track_progress(temperature, iteration)
  endif
```

```
  iteration = iteration+1
```

```
enddo
```



**Done?**



**Calculate**



**Update  
temp  
array and  
find max  
change**



**Output**

# Serial Fortran Code Subroutines

```
subroutine initialize( temperature_last )
  implicit none

  integer, parameter      :: columns=1000
  integer, parameter      :: rows=1000
  integer                 :: i,j

  double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

  temperature_last = 0.0

  !these boundary conditions never change throughout run

  !set left side to 0 and right to linear increase
  do i=0,rows+1
    temperature_last(i,0) = 0.0
    temperature_last(i,columns+1) = (100.0/rows) * i
  enddo

  !set top to 0 and bottom to linear increase
  do j=0,columns+1
    temperature_last(0,j) = 0.0
    temperature_last(rows+1,j) = ((100.0)/columns) * j
  enddo

end subroutine initialize
```

```
subroutine track_progress(temperature, iteration)
  implicit none

  integer, parameter      :: columns=1000
  integer, parameter      :: rows=1000
  integer                 :: i,iteration

  double precision, dimension(0:rows+1,0:columns+1) :: temperature

  print *, '----- Iteration number: ', iteration, ' -----'
  do i=5,0,-1
    write (*, '(("i4,",",i4,"):",f6.2," ")',advance='no'), &
      rows-i,columns-i,temperature(rows-i,columns-i)
  enddo
  print *
```

# Whole Fortran Code

```

program serial
  implicit none

  !Size of plate
  integer, parameter      :: columns=1000
  integer, parameter      :: rows=1000
  double precision, parameter :: max_temp_error=0.01

  integer                :: i, j, max_iterations, iteration=1
  double precision       :: dt=100.0
  real                   :: start_time, stop_time

  double precision, dimension(0:rows+1,0:columns+1) :: temperature, temperature_last

  print*, 'Maximum iterations [100-4000]?'
  read*,   max_iterations

  call cpu_time(start_time)    !Fortran timer

  call initialize(temperature_last)

  !do until error is minimal or until maximum steps
  do while ( dt > max_temp_error .and. iteration <= max_iterations)

    do j=1,columns
      do i=1,rows
        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) )
      enddo
    enddo

    dt=0.0

    !copy grid to old grid for next iteration and find max change
    do j=1,columns
      do i=1,rows
        dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
        temperature_last(i,j) = temperature(i,j)
      enddo
    enddo

    !periodically print test values
    if( mod(iteration,100).eq.0 ) then
      call track_progress(temperature, iteration)
    endif

    iteration = iteration+1

  enddo

  call cpu_time(stop_time)

  print*, 'Max error at iteration ', iteration-1, ' was ',dt
  print*, 'Total time was ',stop_time-start_time, ' seconds.'

end program serial

```

```

! initialize plate and boundary conditions
! temp_last is used to to start first iteration
subroutine initialize( temperature_last )
  implicit none

  integer, parameter      :: columns=1000
  integer, parameter      :: rows=1000
  integer                :: i,j

  double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

  temperature_last = 0.0

  !these boundary conditions never change throughout run

  !set left side to 0 and right to linear increase
  do i=0,rows+1
    temperature_last(i,0) = 0.0
    temperature_last(i,columns+1) = (100.0/rows) * i
  enddo

  !set top to 0 and bottom to linear increase
  do j=0,columns+1
    temperature_last(0,j) = 0.0
    temperature_last(rows+1,j) = ((100.0)/columns) * j
  enddo

end subroutine initialize

!print diagonal in bottom corner where most action is
subroutine track_progress(temperature, iteration)
  implicit none

  integer, parameter      :: columns=1000
  integer, parameter      :: rows=1000
  integer                :: i, iteration

  double precision, dimension(0:rows+1,0:columns+1) :: temperature

  print *, '----- Iteration number: ', iteration, ' -----'
  do i=5,0,-1
    write (*,('("i4,"",",i4,"):" ",f6.2," " )',advance='no'), &
           rows-i,columns-i,temperature(rows-i,columns-i)

    enddo
    print *
  end subroutine track_progress

```

# Exercises: General Instructions for Compiling

- Exercises are in the “Exercises/OpenACC” directory in your home directory
- Solutions are in the “Solutions” subdirectory
- To compile

```
pgcc -acc laplace.c
pgf90 -acc laplace.f90
```
- This will generate the executable **a.out**

# Our Workshop Environment

John Urbanic  
Parallel Computing Scientist  
Pittsburgh Supercomputing Center



# Our Environment This Week

- Your laptops or workstations: only used for portal access
- Bridges is our HPC platform

We will here briefly go through the steps to login, edit, compile and run before we get into the real materials.

We want to get all of the distractions and local trivia out of the way here. Everything *after* this talk applies to any HPC environment you will encounter.

20 Storage Building Blocks, implementing the parallel *Pylon* filesystem (~10PB) using PSC's SLASH2 filesystem

2 MDS nodes  
2 front-end nodes  
2 boot nodes

8 management nodes  
6 "core" Intel® OPA edge switches: fully interconnected, 2 links per switch

Intel® OPA cables

800 HPE Apollo 2000 (128GB) compute nodes

4 HPE Integrity Superdome X (12TB) compute nodes

42 HPE ProLiant DL580 (3 TB) compute nodes  
12 HPE ProLiant DL380 database nodes

6 HPE ProLiant DL360 web server nodes

20 "leaf" Intel® OPA edge switches

32 RSM nodes with NVIDIA next-generation GPUs  
16 RSM nodes with NVIDIA K80 GPUs

Purpose-built Intel® Omni-Path topology for data-intensive HPC

<http://staff.psc.edu/nystrom/bvt>

# Bridges Node Types

Type	RAM <sup>a</sup>	Phse.	n	CPU / GPU / other	Server
ESM	12TB	1	2	16 × Intel Xeon E7-8880 v3 (18c, 2.3/3.1 GHz, 45MB LLC)	HPE Integrity Superdome X
		2	2	16 × TBA	
LSM	3TB	1	8	4 × Intel Xeon E5-8860 v3 (16c, 2.2/3.2 GHz, 40 MB LLC)	HPE ProLiant DL580
		2	34	4 × TBA	
RSM	128GB	1	752	2 × Intel Xeon E5-2695 v3 (14c, 2.3/3.3 GHz, 35MB LLC)	HPE Apollo 2000
RSM-GPU	128GB	1	16	2 × Intel Xeon E5-2695 v3 + 2 × NVIDIA K80	
		2	32	2 × Intel Xeon E5-2695 v3 + 2 × NVIDIA next-generation GPU	
DB-s	128GB	1	6	2 × Intel Xeon E5-2695 v3 + SSD	HPE ProLiant DL360
DB-h			6	2 × Intel Xeon E5-2695 v3 + HDDs	HPE ProLiant DL380
Web	128GB	1	6	2 × Intel Xeon E5-2695 v3	HPE ProLiant DL360
Other <sup>b</sup>	128GB	1	14	2 × Intel Xeon E5-2695 v3	HPE ProLiant DL360, DL380
Total					

a. All RAM in these nodes is DDR4-2133

b. Other nodes = front end (2) + management/log (8) + boot (2) + MDS (4)

# Getting Connected

- The first time you use your account sheet, you must go to [apr.psc.edu](http://apr.psc.edu) to set a password. We will take a minute to do this shortly.
- We will be working on [bridges.psc.edu](http://bridges.psc.edu). Use an ssh client (a Putty terminal, for example), to ssh to the machine.
- At this point you are on a login node. It will have a name like “br001” or “br006”. This is a fine place to edit and compile codes. However we must be on compute nodes to do actual computing. We have designed Bridges to be the world’s most interactive supercomputer. We generally only require you to use the batch system when you want to. Otherwise, you get your own personal piece of the machine. To get a single GPU use “interact -p GPU”:

```
[urbanic@br006 ~]$ interact -p GPU
```

```
[urbanic@gpu016 ~]$
```

- You can tell you are on a GPU node because it has a name like “gpu012”.
- Do make sure you are working on a GPU node. Otherwise your results will be confusing.
- We could request different types of nodes (GPU for OpenACC or many cores for OpenMP, for example). In general, you can use the interact session you request for the rest of the day unless you need to request different resources.

# Editors

For editors, we have several options:

- emacs
- vi
- nano: use this if you aren't familiar with the others

# Compiling

We will be using standard Fortran and C compilers this week. They should look familiar.

- pgcc for C
- pgf90 for Fortran

We will slightly prefer the PGI compilers (the Intel or gcc ones would also be fine for most of our work, but not so much for OpenACC). There are also MPI wrappers for these called mpicc and mpif90 that we will use. Note that on Bridges you would normally have to enable this compiler with

```
module load pgi
```

I have put that in the .bashrc file that we will all start with.

# Multiple Sessions

You are limited to one interactive compute node session for our workshop. However, there is no reason not to open other sessions (windows) to the login nodes for compiling and editing. You may find this convenient. Feel free to do so.

# Our Setup For This Workshop

After you copy the files from the training directory, you will have:

```
/Exercises
  /Test
  /OpenMP
    laplace_serial.f90/c
  /solutions
  /Examples
  /Prime
  /OpenACC
  /MPI
```



# Preliminary Exercise

Let's get the boring stuff out of the way now.

- Log on to `apr.psc.edu` and set an initial password.

- Log on to Bridges.

```
ssh username@bridges.psc.edu
```

- Copy the exercise directory from the training directory to your home directory, and then copy the workshop shell script into your home directory.

```
cp -r ~training/Exercises .  
cp ~training/.bashrc .
```

- Logout and back on again to activate this script. You won't need to do that in the future.
- Edit a file to make sure you can do so. Use emacs, vi or nano (if the first two don't sound familiar).
- Start an interactive session.

```
interact -p GPU
```

- cd into your exercises/test directory and compile (C or Fortran)

```
cd Exercises/Test  
pgcc test.c  
pgf90 test.f90
```

- Run your program

```
a.out
```

(You should get back a message of "Congratulations!")

# Exercises: Very useful compiler option

Adding **-Minfo=accel** to your compile command will give you some very useful information about how well the compiler was able to honor your OpenACC directives.

```
instr009@h2ologin2:~/Test> pgcc -acc -Minfo=accel laplace_bad_acc.c
main:
  71, Generating present_or_copyout(Temperature[1:1000][1:1000])
    Generating present_or_copyin(Temperature_old[0:][0:])
    Generating NVIDIA code
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
    Generating compute capability 3.0 binary
  72, Loop is parallelizable
  73, Loop is parallelizable
    Accelerator kernel generated
    72, #pragma acc loop gang /* blockIdx.y */
    73, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  82, Generating present_or_copyin(Temperature[1:1000][1:1000])
    Generating present_or_copy(Temperature_old[1:1000][1:1000])
    Generating NVIDIA code
    Generating compute capability 1.3 binary
    Generating compute capability 2.0 binary
    Generating compute capability 3.0 binary
  83, Loop is parallelizable
  84, Loop is parallelizable
    Accelerator kernel generated
    83, #pragma acc loop gang /* blockIdx.y */
    84, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
  85, Max reduction generated for dt
```

# Exercises: General Instructions for Running

Make sure you are on a GPU node. The command prompt is your clue.

```
urbanic@gpu006 ~] $ a.out
```

You can compare against the serial code you are starting with to see what performance gains you achieve. You can compile the serial version without any extra flags (just pgcc or pgf90), but run it as per the above. Rename your a.out's to avoid confusion.

# Exercise 1: Using kernels to parallelize the main loops

(About 45 minutes)

Q: Can you get a speedup with just the kernels directives?

1. Edit *laplace\_serial.c/f90*
  1. Maybe copy your intended OpenACC version to *laplace\_acc.c* to start
  2. Add directives where it helps
2. Compile with OpenACC parallelization
  1. `pgcc -acc -Minfo=accel laplace_acc.c` or `pgf90 -acc -Minfo=accel laplace_acc.f90`
  2. Look at your compiler output to make sure you are having an effect
3. Run
  1. `a.out` (Try 4000 iterations if you want a solution that converges to current tolerance)
  2. Serial version for baseline time
  3. Your OpenACC version for performance difference