Rules and Regulations of the 2nd Annual IHPCSS Challenge

Trophy bears no relationship to reality.

General Rules

- Due Thursday midnight (!)
- 4 Nodes of Bridges
- Use any combination of MPI, OpenACC, Python and OpenMP
- How fast can you run a 10K x 10K Laplace code to convergence?

Some Specifics

- Can't change kernel (Must retain two core loops source)
- Can change number of MPI processes (Does not have to be 112 or 4)
- 1 Source File
- 1 Combined Environment/Compile/Submit/Execute script
  - to make it easy for us to run your solutions!
- Mail to d.henty@epcc.ed.ac.uk by deadline
  - Mail a ping earlier if you want to be informed of any developments

Rules For Lawyers

- No libraries
- Don’t mess with timer placement
- ?

Reality Checks

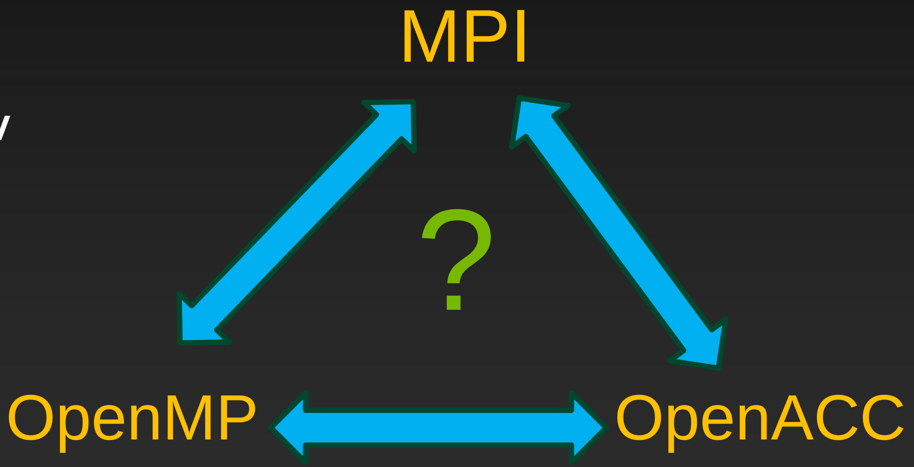
- Serial code converges at 3578 time steps. Yours should too.
- As we know, this is not enough to verify correctness. You should find point [7500][9950] in C and (9950,7500) in Fortran converges to 17 (±1) degrees.
- As discussed, the 10K result differs from the 1K result.*
  - Plugging in Gauss-Seidel or Successive Over Relaxation (SOR) would be easy and interesting. But, not for our contest.

http://www.cs.berkeley.edu/~demmel/cs267/lecture24/lecture24.html is a brief analysis of these issues.

Suggested Things to Explore

Blue Waters User Guide is your friend!
https://bluewaters.ncsa.illinois.edu/user-guide

- Compiler flags
  - -fast
- Compiler
  - see Bridges documentation for how to use different modules
- MPI Environment Variables
  - man mpi
- Thread placement
  - google for KMP_AFFINITY



Decision

- On Thursday evening we will take the top self-reported speeds and run them in an interactive session

- Timings not within 10% of self-reported time will be disqualified

- Codes should print out "test point" at [7500][9950] for C, (9950,7500) for Fortran at conclusion of run.

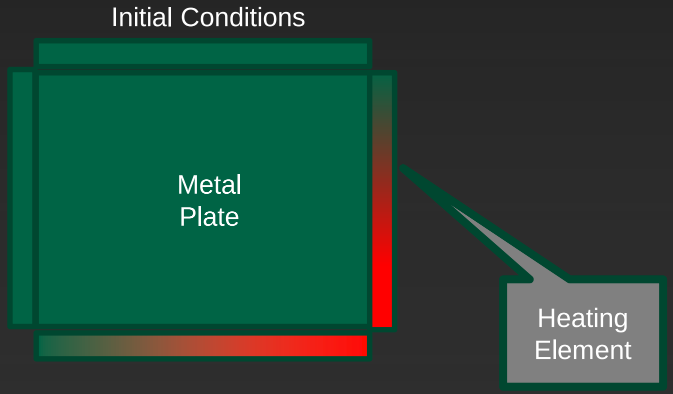
- Best of two runs for each finalist will determine winner

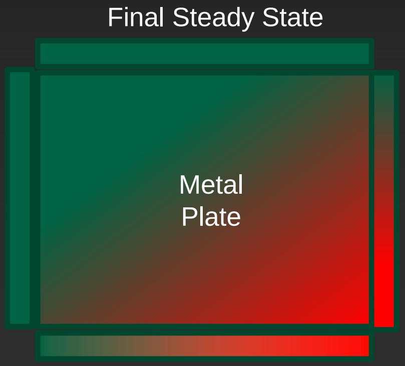
Laplace Exercise

John Urbanic
Parallel Computing Specialist
Pittsburgh Supercomputing Center

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 MPI.
- 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:

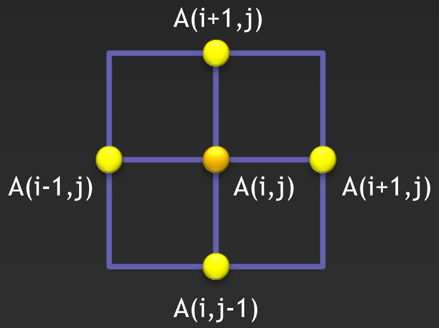




PITTSBURGH SUPERCOMPUTING CENTER

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

```c
#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

PITTSBURGH SUPERCOMPUTING CENTER

Serial Fortran Code Subroutines

```fortran
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
```

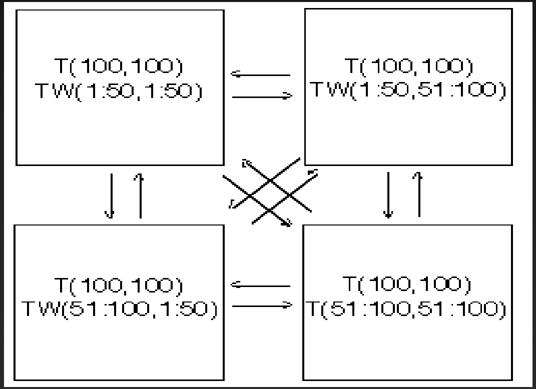
```fortran
subroutine track_progress(temperature, iteration)
      implicit none

      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

```fortran
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
```

```fortran
! initialize plate and boundery 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
```

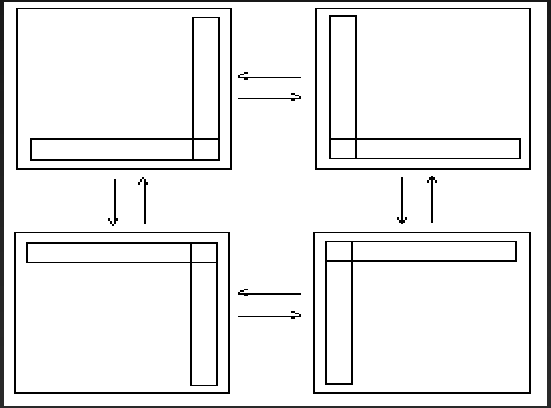
First Things First: Domain Decomposition



- All processors have entire T array.
- Each processor works on TW part of T.
- After every iteration, all processors broadcast their TW to all other processors.
- Increased memory. NOT SCALABLE!
- Global (message passing) variables are ALWAYS bad!

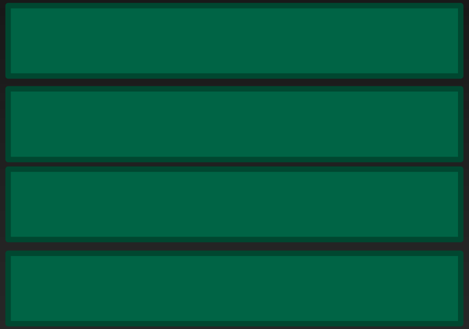
PITTSBURGH SUPERCOMPUTING CENTER

Try Again:
Domain Decomposition II

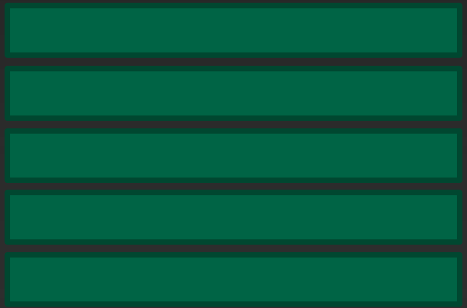


- Each processor has sub-grid.
- Communicate boundary values only.
- Reduces memory.
- Reduces communications.
- Have to keep track of neighbors in two directions.
- But not too bad.

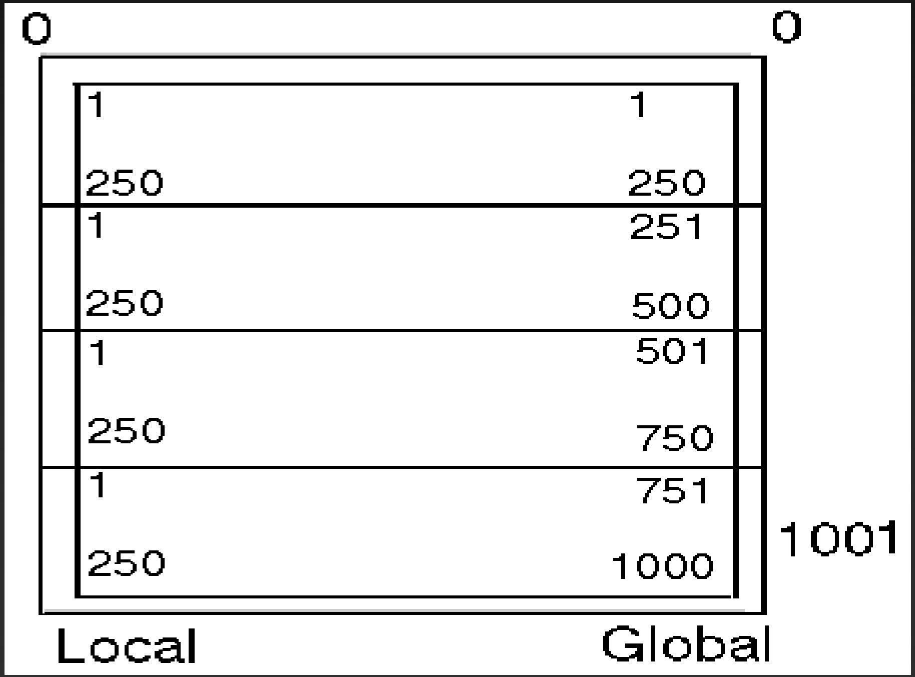
Simplest: Domain Decomposition III



- Only have to keep track of up/down neighbors, and no corner case.
- Scales, as below. How would we handle 5 PEs with the “square decomposition”?

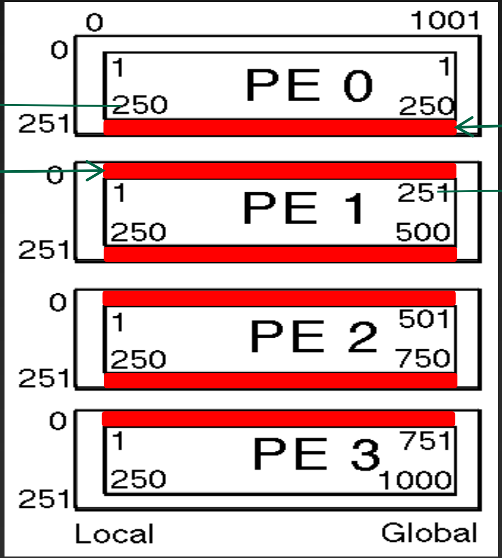


Simplest Decomposition for C Code



Simplest Decomposition for C Code

In the parallel case, we will break this up into 4 processors. There is only one set of boundary values. But when we distribute the data, each processor needs to have an extra row for data distribution, these are commonly called the "ghost cells".



The program has a local view of data. The programmer has to have a global view of data. The ghost cells don't exist in the global dataset. They are only copies from the "real" data in the adjacent PE.

Sending Multiple Elements

- For the first time we want to send multiple elements. In this case, a whole row or column of data. That is exactly what the count parameter is for.

- The common use of the count parameter is to point the Send or Receive routine at the first element of an array, and then the count will proceed to strip off as many elements as you specify.

- This implies (and demands) that the elements are contiguous in memory. That will be true for one dimension of an array, but the other dimension(s) will have a stride.

- In C this is true for our rows. In Fortran this is true for our columns. This will give us a strong preference for the problem orientation in each language. Then we don't have to worry about strides in the strips that we send.

- However, it is often necessary to send messages that are not contiguous data. Using defined data types, we can send other array dimensions, or even blocks or surfaces. We will talk about that capability in the Advanced talk.

PITTSBURGH SUPERCOMPUTING CENTER

Sending Multiple Elements

This last index is the one contiguous in memory.

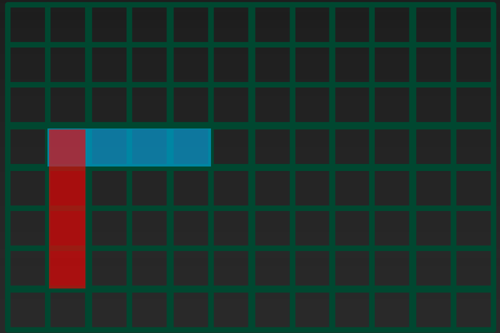
C:

```
int A[8][12];
MPI_Send(&A[3][1], 4, MPI_INT, pe, tag, MPI_COMM_WORLD);
```

Fortran:

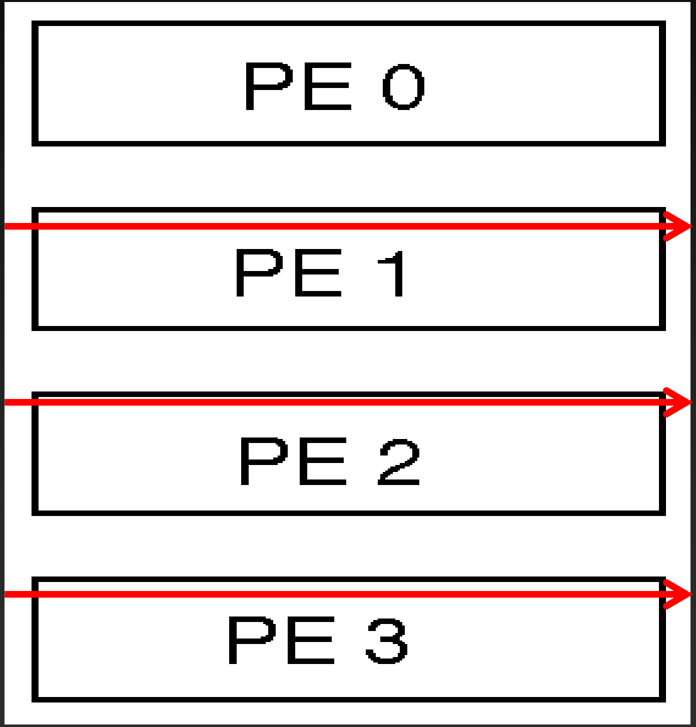
```
integer A(0:7,0:11)
MPI_Send(A(3,1), 4, MPI_INT, pe, tag, MPI_COMM_WORLD, err);
```

This first index is the one contiguous in memory.

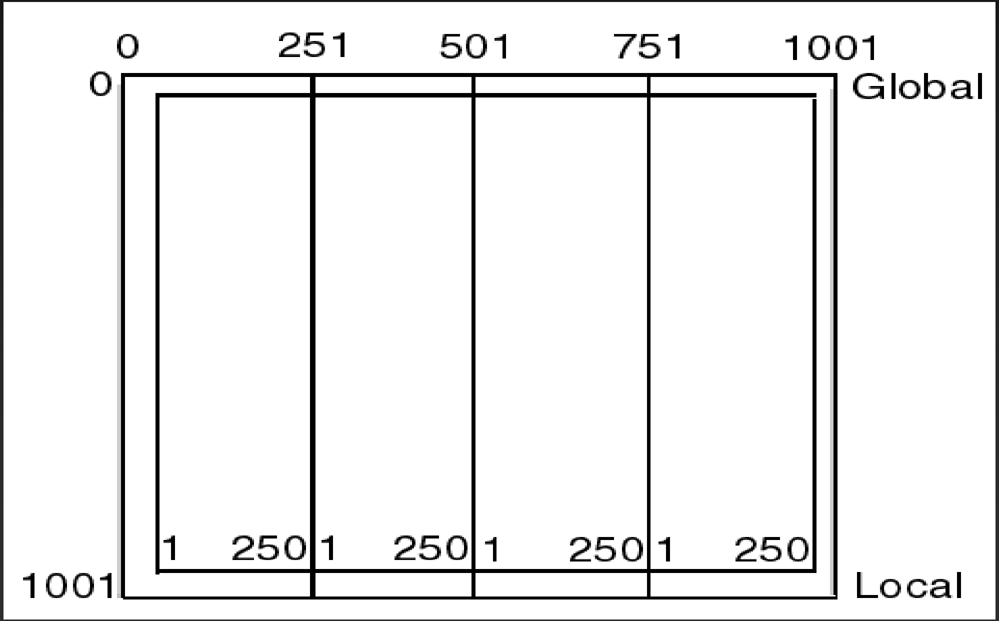


Sending Multiple Elements

```
if ( mype != 0 ){
    up = mype - 1
    MPI_Send( t, COLUMNS, MPI_FLOAT, up, UP_TAG, comm);
}

Alternatively
up = mype - 1
if ( mype == 0 ) up = MPI_PROC_NULL;
MPI_Send( t, COLUMNS, MPI_FLOAT, up, UP_TAG, comm);
```

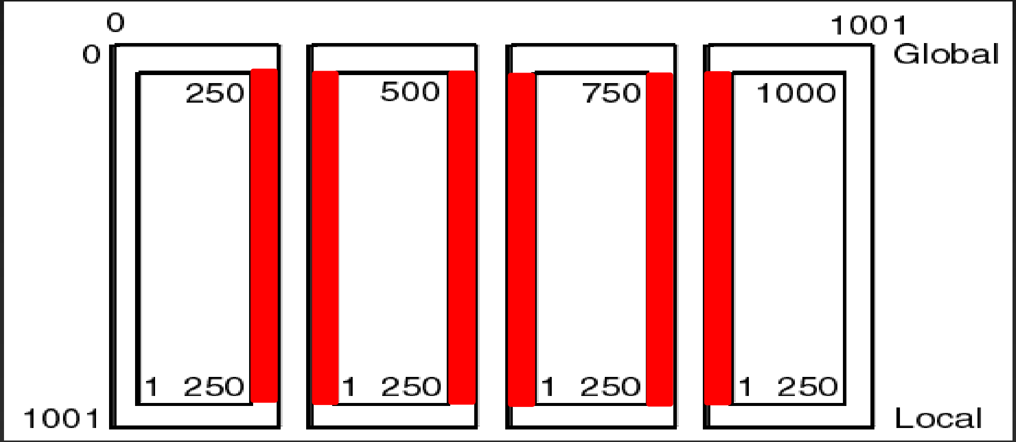


Simplest Decomposition for Fortran Code



Simplest Decomposition for Fortran Code

Then we send strips to ghost zones like this:



Same ghost cell structure as the C code, we have just swapped rows and columns.

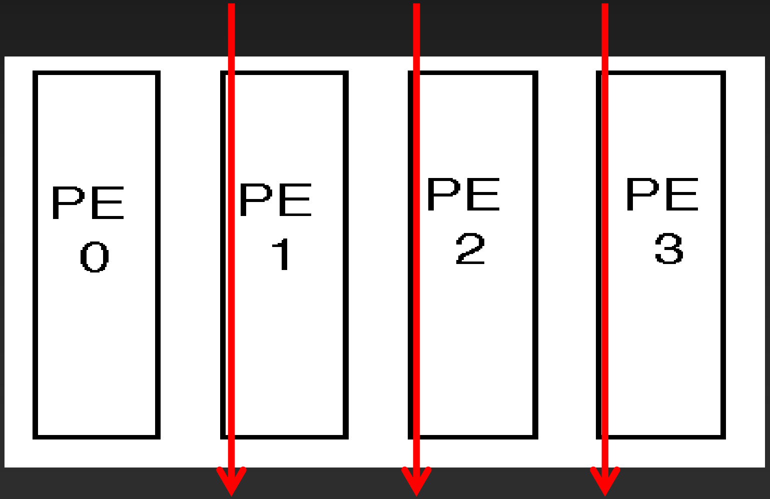
Sending Multiple Elements in Fortran

```
if( mype.ne.0 ) then
    left = mype - 1
    call MPI_Send( t, ROWS, MPI_REAL, left, L_TAG, comm, ierr)
endif
```

Alternatively

```
left = mype - 1
if( mype.eq.0 ) left = MPI_PROC_NULL
call MPI_Send( t, ROWS, MPI_REAL, left, L_TAG, comm, ierr)
endif
```

Note: You may also `MPI_Recv from MPI_PROC_NULL`



Main Loop Structure

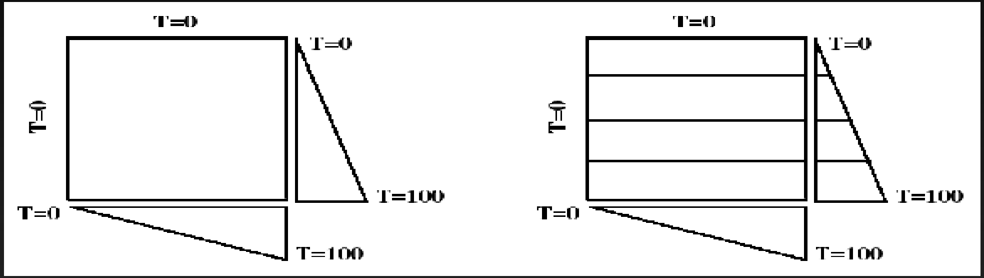
```
for (iter=1; iter < NITER; iter++) {
    Do averaging
    Copy Temperature into Temperature_last
```

Compute Phase
(almost unchanged)

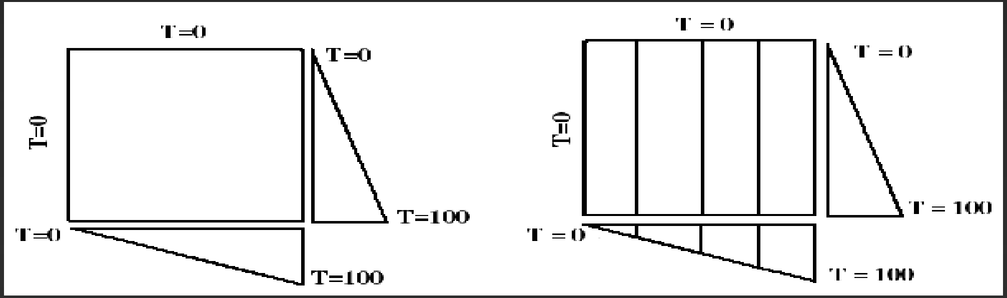
```
Send real values down
```
Temperature or Temperature_last ?
```
Send real values up

Receive values from above into ghost zone

Receive values from below into ghost zone
```
Temperature or Temperature_last?
```
Find the max change

Synchronize?
```

Communicate Phase
(all new)

Boundary Conditions



Both C and Fortran will need to set proper boundary conditions based upon the PE number.



Two ways to approach this exercise.

- Start from the serial code
- Start from the template ("hint") code

Staring files in /MPI:

```
laplace_serial.c          laplace_serial.f90
laplace_template.c        laplace_template.f90
```

You can peek at my answer in /Solutions

```
laplace_mpi.c             laplace_mpi.f90
```

MPI Template for C

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

    int i, j;
    int max_iterations;
    int iteration=1;

    // the usual MPI startup routines
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    // verify only NPES PEs are being used
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    // PE 0 asks for input
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    // bcast max iterations to other PEs
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

    if (my_PE_num==0) gettimeofday(&start_time,NULL);

    initialize(npes, my_PE_num);

    while ( dt_global > 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]);
            }
        }

        // COMMUNICATION PHASE: send and receive ghost rows for next iteration
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

        dt = 0.0;
.
.
.
```

PITTSBURGH SUPERCOMPUTING CENTER

MPI Template for Fortran

```
.
.
program mpi
      implicit none
      include    'mpif.h'

      !Size of plate
      integer, parameter             :: columns_global=1000
      integer, parameter             :: rows=1000

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

      !usual mpi startup routines
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      !It is nice to verify that proper number of PEs are running
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      !Only one PE should prompt user
      if( mype == 0 ) then
         print*, 'Maximum iterations [100-4000]?'
         read*,   max_iterations
      endif

      !Other PEs need to recieve this information
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>

      call cpu_time(start_time)

      call initialize(temperature_last, npes, mype)

      !do until global error is minimal or until maximum steps
      do while ( dt_global > 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
```

Some ways we might get fancy…

Send and receive at the same time:

```
MPI_Sendrecv( … )
```

Defined Data Types:

```
MPI_Datatype row, column ;
MPI_Type_vector ( COLUMNS, 1, 1, MPI_DOUBLE, & row );
MPI_Type_vector ( ROWS, 1, COLUMNS, MPI_DOUBLE , & column );
MPI_Type_commit ( & row );
MPI_Type_commit ( & column );
.
.
//Send top row to up neighbor (what we've been doing)
MPI_Send(Temperature[1,1], 1, row, dest, tag, MPI_COMM_WORLD);
//Send last column to right hand neighbor (in a new 2D layout)
MPI_Send(Temperature[1,COLUMNS], 1, column, dest, tag, MPI_COMM_WORLD);
```

PITTSBURGH SUPERCOMPUTING CENTER

Some ways you might go wrong…

You have two main data structures

- Temperature
- Temperature_last

Each has

- Boundary Conditions (unchanged through entire run)
- Ghost zones (changing every timestep)

Each iteration

- Copying/calculating Temperature to/from Temperature_last
- Sending/receiving into/from ghost zones and data

It is easy to mix these things up. I suggest you step through at least the initialization and first time step for each of the above combinations of elements.

There are multiple reasonable solutions. Each will deal with the above slightly differently.

PITTSBURGH SUPERCOMPUTING CENTER

How do you know you are correct?

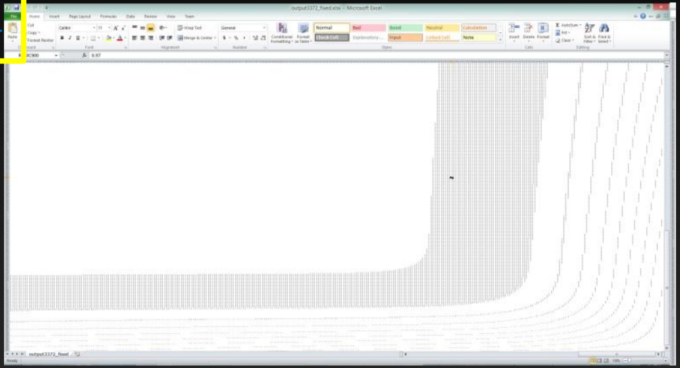
Your solution converges
at 3372 timesteps!

How do you know you are correct?

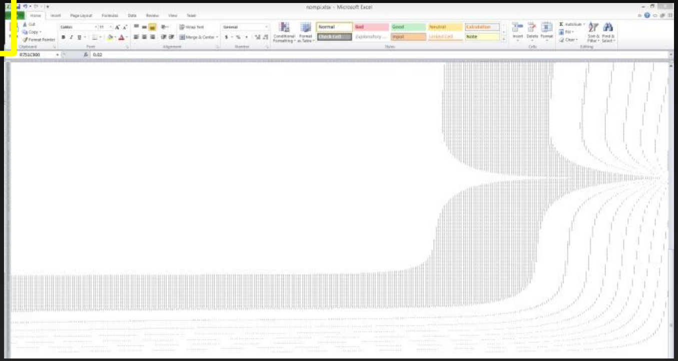
Your solution converges
at 3372 timesteps!

How do you know you are correct?

Bottom Right Corner



Bottom Right Corner



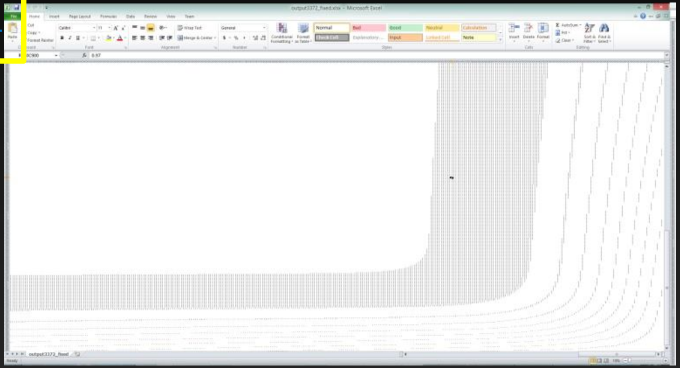
Working MPI Solution

MPI Routines Disabled

Both converge at 3372 steps!

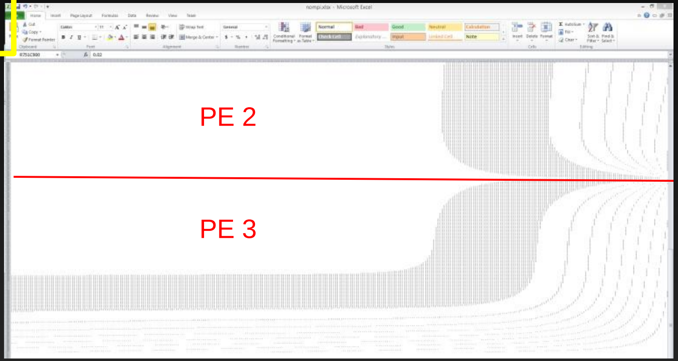
How do you know you are correct?

Bottom Right Corner



Working MPI Solution

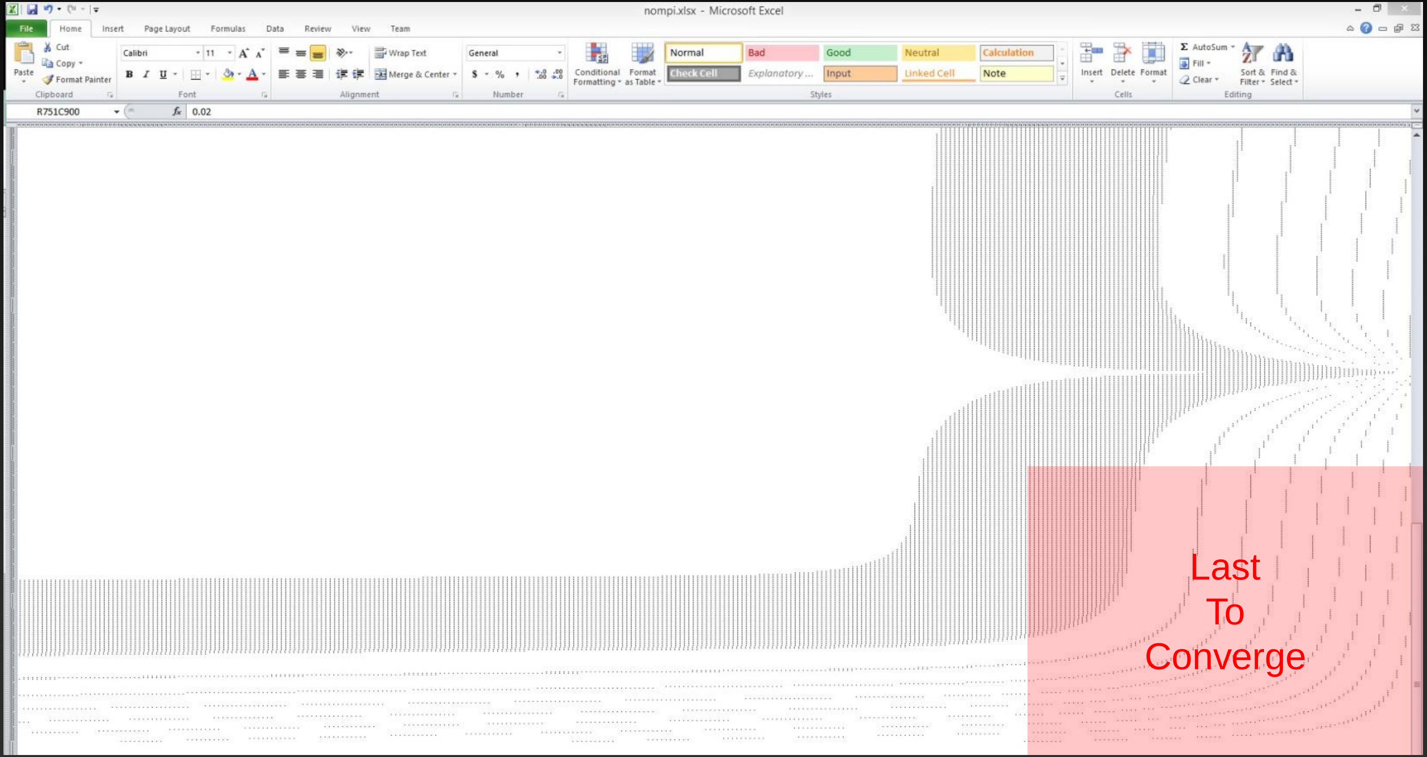
Bottom Right Corner



MPI Routines Disabled

Both converge at 3372 steps!

All the action is here.



Check for yourself.

```
void output(int my_pe, int iteration) {

  FILE* fp;
  char filename[50];

  sprintf(filename,"output%d.txt",iteration);

  for (int pe = 0; pe<4; pe++){
    if (my_pe==pe){

      fp = fopen(filename, "a");

      for(int y = 1; y <= ROWS; y++){
        for(int x = 1; x <= COLUMNS; x ++){
          fprintf(fp, "%5.2f ",Temperature[y][x]);
        }
        fprintf(fp,"\n");
      }

      fflush(fp);
      fclose(fp);

    }
    MPI_Barrier(MPI_COMM_WORLD);

  }

}
```

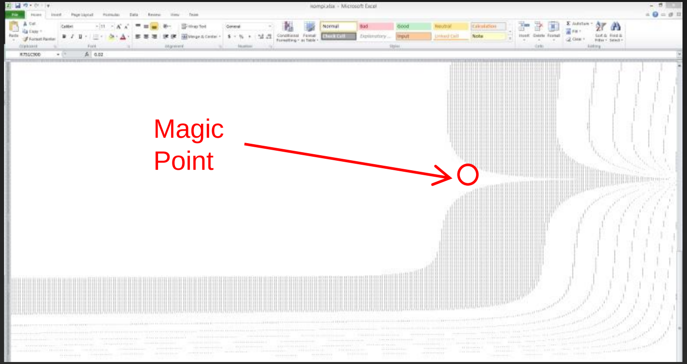
- Human Readable
- 1M entries
- Visualize. I used Excel (terrible idea).

Check for yourself.

```
void output(int my_pe, int iteration) {

  FILE* fp;
  char filename[50];

  sprintf(filename,"output%d.txt",iteration);

  for (int pe = 0; pe<4; pe++){
    if (my_pe==pe){

      fp = fopen(filename, "a");

      for(int y = 1; y <= ROWS; y++){
        for(int x = 1; x <= COLUMNS; x ++){
          fprintf(fp, "%5.2f ",Temperature[y][x]);
        }
        fprintf(fp,"\n");
      }

      fflush(fp);
      fclose(fp);

    }
    MPI_Barrier(MPI_COMM_WORLD);

  }

}
```

C:
```
if (my_PE_num==2)
      printf("Global coord [750,900] is %f \n:", Temperature[250][900]);
```

Fortran:
```
if (mype==2) then
         print*, 'magic point', temperature(900,250)
endif
```



- Human Readable
- 1M entries
- Visualize. I used Excel (terrible idea).

- If about 1.0, probably good
- Otherwise (like 0.02 here) probably not

Laplace Exercise

1. You copied a directory called MPI_Course/Laplace into your home directory. Go there and you will see the files:

```
laplace_template.c and laplace_serial.c
or
laplace_template.f90 and laplace_serial.f90
```

2. The templates are "hint" files with sections marked >>>>> in the source code where you might add statements so that the code will run on 4 PEs. You can start from either these or from the serial code, whichever you prefer. A useful Web reference for this exercise is the Message Passing Interface Standard at:

   http://www.mpich.org/static/docs/v3.0.x/

3. To compile the program as it becomes an MPI code, execute:

```
mpicc laplace_your_mpi.c
mpif90 laplace_your_mpi.f90
```

4. In an interactive idev session, you can just run these as:

```
ibrun -np 4 a.out
```

5. You can check your program against one possible solution in the Solutions directory:

```
laplace_mpi.c  or  laplace_mpi.f90
```

6. When you are done, let us know by hitting the survey button on the workshop page: bit.ly/XSEDE-Workshop

Introduction to OpenMP

Lecture 6: Further topics in OpenMP

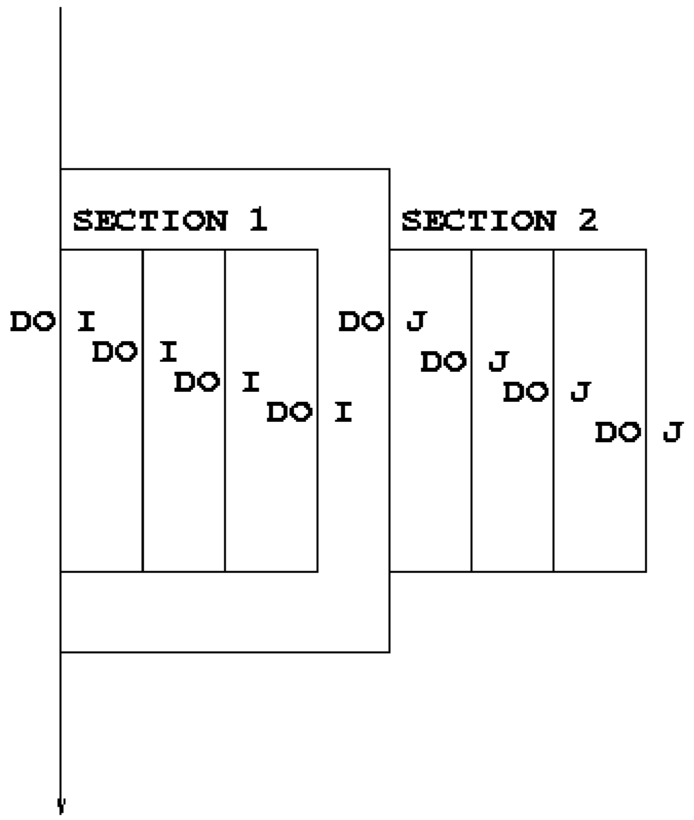
Nested parallelism

- Unlike most previous directive systems, nested parallelism is permitted in OpenMP.
- This is enabled with the **`OMP_NESTED`** environment variable or the **`OMP_SET_NESTED`** routine.
- If a PARALLEL directive is encountered within another PARALLEL directive, a new team of threads will be created.
- The new team will contain only one thread unless nested parallelism is enabled.

Nested parallelism (cont)

Example:

```
!$OMP PARALLEL
!$OMP SECTIONS
!$OMP SECTION
!$OMP PARALLEL DO
      do i = 1,n
         x(i) = 1.0
      end do
!$OMP SECTION
!$OMP PARALLEL DO
      do j = 1,n
         y(j) = 2.0
      end do
!$OMP END SECTIONS
!$OMP END PARALLEL
```



Nested parallelism (cont)

- Not often needed, but can be useful to exploit non-scalable parallelism (SECTIONS).

- Note: nested parallelism isn’t supported in some implementations (the code will execute, but as if OMP_NESTED is set to FALSE).
  - turns out to be hard to do correctly without impacting performance significantly.

NUMTHREADS clause

- One way to control the number of threads used at each level is with the NUM_THREADS clause:

```
!$OMP PARALLEL DO NUM_THREADS(4)
      DO I = 1,4
!$OMP PARALLEL DO NUM_THREADS(TOTALTHREADS/4)
         DO J = 1,N
            A(I,J) = B(I,J)
         END DO
      END DO
```

- The value set in the clause supersedes the value in the environment variable OMP_NUM_THREADS (or that set by **`omp_set_num_threads()`**)

Orphaned directives

- Directives are active in the *dynamic* scope of a parallel region, not just its *lexical* scope.
- Example:

```
!$OMP PARALLEL
      call fred()
!$OMP END PARALLEL

      subroutine fred()
!$OMP DO
      do i = 1,n
         a(i) = a(i) + 23.5
      end do
      return
      end
```

Orphaned directives (cont)

- This is very useful, as it allows a modular programming style….
- But it can also be rather confusing if the call tree is complicated (what happens if **`fred`** is also called from outside a parallel region?)
- There are some extra rules about data scope attributes….

Data scoping rules

When we call a subroutine from inside a parallel region:

- Variables in the argument list inherit their data scope attribute from the calling routine.
- Global variables in C++ and COMMON blocks or module variables in Fortran are shared, unless declared THREADPRIVATE (see later).
- **`static`** local variables in C/C++ and **`SAVE`** variables in Fortran are shared.
- All other local variables are private.

Binding rules

- There could be ambiguity about which parallel region directives refer to, so we need a rule….

- DO/FOR, SECTIONS, SINGLE, MASTER and BARRIER directives always bind to the nearest enclosing PARALLEL directive.

Thread private global variables

- It can be convenient for each thread to have its own copy of variables with global scope (e.g. COMMON blocks and module data in Fortran, or file-scope and namespace-scope variables in C/C++).

- Outside parallel regions and in MASTER directives, accesses to these variables refer to the master thread's copy.

Thread private globals (cont)

Syntax:

Fortran: **`!$OMP THREADPRIVATE (`***list***`)`**

where list contains named common blocks (enclosed in slashes), module variables and SAVEd variables..

This directive must come after all the declarations for the common blocks or variables.

C/C++: **`#pragma omp threadprivate (`***list***`)`**

This directive must be at file or namespace scope, after all declarations of variables in *list* and before any references to variables in *list*. See standard document for other restrictions.

COPYIN clause

- Allows the values of the master thread's THREADPRIVATE data to be copied to all other threads at the start of a parallel region.

Syntax:

Fortran: `COPYIN(`*list*`)`

C/C++: `copyin(`*list*`)`

In Fortran the list can contain variables in THREADPRIVATE COMMON blocks.

COPYIN clause

Example:

```
      common /junk/ nx
      common /stuff/ a,b,c
!$OMP THREADPRIVATE (/JUNK/,/STUFF/)
      nx = 32
      c = 17.9
      . . .
!$OMP PARALLEL PRIVATE(NX2,CSQ) COPYIN(/JUNK/,C)
      nx2 = nx * 2
      csq = c*c
      . . .
```

Timing routines

OpenMP supports a portable timer:

– return current wall clock time (relative to arbitrary origin) with:

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

```
double omp_get_wtime(void);
```

– return clock precision with

```
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
```

```
double omp_get_wtick(void);
```

Using timers

```
DOUBLE PRECISION STARTTIME, TIME

STARTTIME = OMP_GET_WTIME()
......(work to be timed)
TIME = OMP_GET_WTIME()- STARTTIME
```

Note: timers are local to a thread: must make both calls on the same thread.

Also note: no guarantees about resolution!

Exercise

Molecular dynamics again

- Aim: use of orphaned directives.
- Modify the molecular dynamics code so by placing a parallel region directive around the iteration loop in the main program, and making all code within this sequential except for the forces loop.
- Modify the code further so that each thread accumulates the forces into a local copy of the force array, and reduce these copies into the main array at the end of the loop.

Introduction to OpenMP

Lecture 7: Tasks

OpenMP tasks

- The task construct defines a section of code
- Inside a parallel region, a thread encountering a task construct will package up the task for execution
- Some thread in the parallel region will execute the task at some point in the future

`task` directive

Syntax:

Fortran:

```
!$OMP TASK [clauses]
       structured block
!$OMP END TASK
```

C/C++:

```
#pragma omp task [clauses]
    structured-block
```

Data Sharing

- The default for tasks is usually firstprivate, because the task may not be executed until later (and variables may have gone out of scope).
- Variables that are shared in all constructs starting from the innermost enclosing parallel construct are shared.

```
#pragma omp parallel shared(A) private(B)
{
   ...
#pragma omp task
   {
      int C;
      compute(A, B, C);
   }
}
```

A is shared
B is firstprivate
C is private

When/where are tasks complete?

- At thread barriers (explicit or implicit)
  - applies to all tasks generated in the current parallel region up to the barrier

- At taskwait directive
  - i.e. Wait until all tasks defined in the current task have completed.
  - Fortran: `!$OMP TASKWAIT`
  - C/C++: `#pragma omp taskwait`

  - Note: applies only to tasks generated in the current task, not to “descendants” .

Example

```
p = listhead ;
while (p) {
  process (p);
  p=next(p) ;
}
```

- Classic linked list traversal
- Do some work on each item in the list
- Assume that items can be processed independently
- Cannot use an OpenMP loop directive

Parallel pointer chasing

Only one thread packages tasks

```
#pragma omp parallel
{
  #pragma omp single private(p)
   {
    p = listhead ;
    while (p) {
       #pragma omp task
               process (p);
       p=next (p) ;
     }
   }
}
```

`p` is firstprivate by default inside this task

Parallel pointer chasing on multiple lists

```
#pragma omp parallel
{
   #pragma omp for private(p)
   for ( int i =0; i <numlists ; i++) {
        p = listheads [ i ] ;
        while (p ) {
        #pragma omp task
            process (p);
        p=next (p ) ;
        }
   }
}
```

All threads package tasks

Example: postorder tree traversal

- Binary tree of tasks
- Traversed using a recursive function
- A task cannot complete until all tasks below it in the tree are complete

```
void postorder(node *p) {
    if (p->left)
      #pragma omp task
        postorder(p->left);
    if (p->right)
      #pragma omp task
        postorder(p->right);
    #pragma omp taskwait
    process(p->data);
}
```

Parent task suspended until children tasks complete

Task switching

- Certain constructs have task scheduling points at defined locations within them
- When a thread encounters a task scheduling point, it is allowed to suspend the current task and execute another (called *task switching*)
- It can then return to the original task and resume

Task switching

```
#pragma omp single
{
  for (i=0; i<ONEZILLION; i++)
    #pragma omp task
      process(item[i]);
}
```

- Risk of generating too many tasks
- Generating task will have to suspend for a while
- With task switching, the executing thread can:
  - execute an already generated task (draining the “*task pool*”)
  - execute the encountered task

Using tasks

- Getting the data attribute scoping right can be quite tricky
  - default scoping rules different from other constructs
  - as ever, using `default(none)` is a good idea

- Don’t use tasks for things already well supported by OpenMP
  - e.g. standard do/for loops
  - the overhead of using tasks is greater

- Don’t expect miracles from the runtime
  - best results usually obtained where the user controls the number and granularity of tasks

Exercise

- Mandelbrot example using tasks.

Advanced OpenMP

Lecture 4: OpenMP and MPI

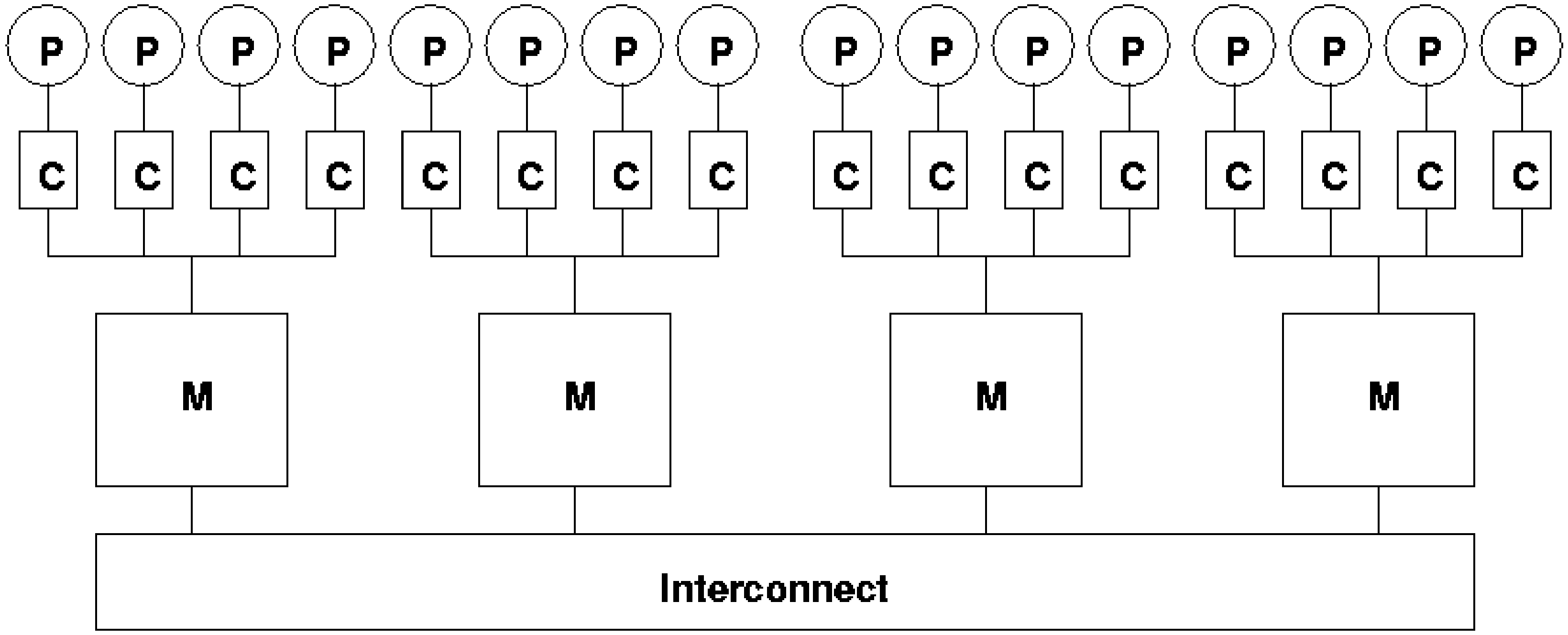
Motivation

- In recent years there has been a trend towards *clustered architectures*

- Distributed memory systems, where each node consist of a traditional shared memory multiprocessor (SMP).
  - with the advent of multicore chips, every cluster is like this

- Single address space within each node, but separate nodes have separate address spaces.

Clustered architecture


P P P P P P P P P P P P P P P P
C C C C C C C C C C C C C C C C
M M M M
Interconnect

Programming clusters

- How should we program such a machine?
- Could use MPI across whole system
- Cannot (in general) use OpenMP/threads across whole system
  - requires support for single address space
  - this is possible in software, but inefficient
  - also possible in hardware, but expensive
- Could use OpenMP/threads within a node and MPI between nodes
  - is there any advantage to this?

Issues

We need to consider:

- Development / maintenance costs
- Portability
- Performance

Development / maintenance

- In most cases, development and maintenance will be harder than for an MPI code, and much harder than for an OpenMP code.

- If MPI code already exists, addition of OpenMP may not be *too* much overhead.

- In some cases, it may be possible to use a simpler MPI implementation because the need for scalability is reduced.
  - e.g. 1-D domain decomposition instead of 2-D

Portability

- Both OpenMP and MPI are themselves highly portable (but not perfect).
- Combined MPI/OpenMP is less so
  - main issue is thread safety of MPI
  - if maximum thread safety is assumed, portability will be reduced
- Desirable to make sure code functions correctly (maybe with conditional compilation) as stand-alone MPI code (and as stand-alone OpenMP code?)

Thread Safety

- Making libraries thread-safe can be difficult
  - lock access to data structures
  - multiple data structures: one per thread
  - …

- Adds significant overheads
  - which may hamper standard (single-threaded) codes

- MPI defines various classes of thread usage
  - library can supply an appropriate implementation
  - see later

Performance

Four possible performance reasons for mixed OpenMP/MPI codes:

1. Replicated data
2. Poorly scaling MPI codes
3. Limited MPI process numbers
4. MPI implementation not tuned for SMP clusters

Replicated data

- Some MPI codes use a replicated data strategy
  - all processes have a copy of a major data structure
  - classical domain decomposition code have replication in halos
  - MPI buffers can consume significant amounts of memory
- A pure MPI code needs one copy per process/core.
- A mixed code would only require one copy per node
  - data structure can be shared by multiple threads within a process
  - MPI buffers for intra-node messages no longer required
- Will be increasingly important
  - amount of memory per core is not likely to increase in future
- Halo regions are a type of replicated data
  - can become significant for small domains (i.e. many processes)

Effect of domain size on halo storage

- Typically, using more processors implies a smaller domain size per processor
  - unless the problem can genuinely weak scale
- Although the amount of halo data does decrease as the local domain size decreases, it eventually starts to occupy a significant amount fraction of the storage
  - even worse with deep halos or >3 dimensions

Local domain size	**Halos**	**% of data in halos**
$50^3 = 125000$	$52^3 – 50^3 = 15608$	11%
$20^3 = 8000$	$22^3 – 20^3 = 2648$	25%
$10^3 = 1000$	$12^3 – 10^3 = 728$	42%

Poorly scaling MPI codes

- If the MPI version of the code scales poorly, then a mixed MPI/OpenMP version *may* scale better.
- May be true in cases where OpenMP scales better than MPI due to:

1. Algorithmic reasons.
   - e.g. adaptive/irregular problems where load balancing in MPI is difficult.
2. Simplicity reasons
   - e.g. 1-D domain decomposition

Load balancing

- Load balancing between MPI processes can be hard
  - need to transfer both computational tasks and data from overloaded to underloaded processes
  - transferring small tasks may not be beneficial
  - having a global view of loads may not scale well
  - may need to restrict to transferring loads only between neighbours
- Load balancing between threads is much easier
  - only need to transfer tasks, not data
  - overheads are lower, so fine grained balancing is possible
  - easier to have a global view
- For applications with load balance problems, keeping the number of MPI processes small can be an advantage

Limited MPI process numbers

- MPI library implementation may not be able to handle millions of processes adequately.
  - e.g. limited buffer space
  - Some MPI operations are hard to implement without O(p) computation, or O(p) storage in one or more processes
  - e.g. AlltoAllv, matching wildcards

- Likely to be an issue on very large systems.

- Mixed MPI/OpenMP implementation will reduce number of MPI processes.

MPI implementation not tuned for SMP clusters

- Some MPI implementations are not well optimised for SMP clusters
  - less of a problem these days
- Especially true for collective operations (e.g. reduce, alltoall)
- Mixed-mode implementation naturally does the right thing
  - reduce within a node via OpenMP reduction clause
  - then reduce across nodes with MPI_Reduce
- Mixed-mode code also tends to aggregate messages
  - send one large message per node instead of several small ones
  - reduces latency effects, and contention for network injection

Styles of mixed-mode programming

- Master-only
  - all MPI communication takes place in the sequential part of the OpenMP program (no MPI in parallel regions)
- Funneled
  - all MPI communication takes place through the same (master) thread
  - can be inside parallel regions
- Serialized
  - only one thread makes MPI calls at any one time
  - distinguish sending/receiving threads via MPI tags or communicators
  - be very careful about race conditions on send/recv buffers etc.
- Multiple
  - MPI communication simultaneously in more than one thread
  - some MPI implementations don’t support this
  - …and those which do mostly don’t perform well

OpenMP Master-only

Fortran

```
!$OMP parallel
 work…
!$OMP end parallel

call MPI_Send(…)


!$OMP parallel
 work…
!$OMP end parallel
```

C

```
#pragma omp parallel
{
   work…
}
ierror=MPI_Send(…);
#pragma omp parallel
{
   work…
}
```

OpenMP Funneled

Fortran

```
!$OMP parallel
… work
!$OMP barrier
!$OMP master
  call MPI_Send(…)
!$OMP end master
!$OMP barrier
.. work
!$OMP end parallel
```

C

```
#pragma omp parallel
{
  … work
  #pragma omp barrier
  #pragma omp master
  {
    ierror=MPI_Send(…);
  }
 #pragma omp barrier
  … work
}
```

OpenMP Serialized

Fortran

```
!$OMP parallel
… work
!$OMP critical
  call MPI_Send(…)
!$OMP end critical
… work
!$OMP end parallel
```

C

```
#pragma omp parallel
{
  … work
  #pragma omp critical
  {
    ierror=MPI_Send(…);
  }
  … work
}
```

OpenMP Multiple

Fortran

```
!$OMP parallel
… work
call MPI_Send(…)
… work
!$OMP end parallel
```

C

```
#pragma omp parallel
{
  … work
   ierror=MPI_Send(…);
  … work
}
```

MPI_Init_thread

- MPI_Init_thread works in a similar way to MPI_Init by initialising MPI on the main thread.
- It has two integer arguments:
  - Required ([in] Level of desired thread support)
  - Provided ([out] Level of provided thread support)

- C syntax

```
int MPI_Init_thread(int *argc, char *((*argv)[]), int
   required, int *provided);
```

- Fortran syntax

```
MPI_INIT_THREAD(REQUIRED, PROVIDED, IERROR)
  INTEGER REQUIRED, PROVIDED, IERROR
```

MPI_Init_thread

- MPI_THREAD_SINGLE
  - Only one thread will execute.
- MPI_THREAD_FUNNELED
  - The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread).
- MPI_THREAD_SERIALIZED
  - The process may be multi-threaded, and multiple threads may make MPI calls, but only one at a time: MPI calls are not made concurrently from two distinct threads (all MPI calls are serialized).
- MPI_THREAD_MULTIPLE
  - Multiple threads may call MPI, with no restrictions.

MPI_Init_thread

- These integer values are monotonic; i.e.,
  - MPI_THREAD_SINGLE < MPI_THREAD_FUNNELED < MPI_THREAD_SERIALIZED < MPI_THREAD_MULTIPLE
- Note that these values do not strictly map on to the four MPI/OpenMP Mixed-mode styles as they are more general (i.e. deal with Posix threads where we don't have "parallel regions", etc.)
  - e.g. no distinction here between Master-only and Funneled
  - see MPI standard for full details

MPI_Query_thread()

- MPI_Query_thread() returns the current level of thread support
  - Has one integer argument: provided [in] as defined for MPI_Init_thread()

- C syntax

```
int MPI_query_thread(int *provided);
```

- Fortran syntax

```
MPI_QUERY_THREAD(PROVIDED, IERROR)
   INTEGER PROVIDED, IERROR
```

- Need to compare the output manually, i.e.

```
If (provided < requested) {
   printf("Not a high enough level of thread support!\n");
   MPI_Abort(MPI_COMM_WORLD,1)
     …etc.
}
```

Pitfalls

- The OpenMP implementation may introduce additional overheads not present in the MPI code (e.g. synchronisation, false sharing, sequential sections).
- The mixed implementation may require more synchronisation than a pure OpenMP version, if non-thread-safety of MPI is assumed.
- Implicit point-to-point synchronisation may be replaced by (more expensive) barriers.
- In the pure MPI code, the intra-node messages will often be naturally overlapped with inter-node messages
  - harder to overlap inter-thread communication with inter-node messages.
- NUMA effects can limit the scalability of OpenMP: it may be advantageous to run one MPI process per NUMA domain, rather than one MPI process per node.
  - process placement becomes very important

Master-only

- Advantages
  - simple to write and maintain
  - clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
  - no concerns about synchronising threads before/after sending messages
- Disadvantages
  - threads other than the master are idle during MPI calls
  - all communicated data passes through the cache where the master thread is executing.
  - inter-process and inter-thread communication do not overlap.
  - only way to synchronise threads before and after message transfers is by parallel regions which have a relatively high overhead.
  - packing/unpacking of derived datatypes is sequential.

Example

```
DO I=1,N
   A(I) = B(I) + C(I)
END DO

CALL MPI_BSEND(A(N),1,.....)
CALL MPI_RECV(A(0),1,.....)


DO I = 1,N
   D(I) = A(I-1) + A(I)
END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
    DO I=1,N

       A(I) = B(I) + C(I)

    END DO


    CALL MPI_BSEND(A(N),1,.....)

    CALL MPI_RECV(A(0),1,.....)



    DO I = 1,N

       D(I) = A(I-1) + A(I)

    END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
     DO I=1,N * nthreads

        A(I) = B(I) + C(I)

     END DO


     CALL MPI_BSEND(A(N),1,.....)

     CALL MPI_RECV(A(0),1,.....)




     DO I = 1,N

        D(I) = A(I-1) + A(I)

     END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
     DO I=1,N * nthreads
        A(I) = B(I) + C(I)
     END DO



     CALL MPI_RECV(A(0),1,.....)



     DO I = 1,N
        D(I) = A(I-1) + A(I)
     END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
     DO I=1,N * nthreads

        A(I) = B(I) + C(I)

     END DO


     CALL MPI_BSEND(A(N * nthreads),1,.....)

     CALL MPI_RECV(A(0),1,.....)



     DO I = 1,N

        D(I) = A(I-1) + A(I)

     END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
     DO I=1,N * nthreads
        A(I) = B(I) + C(I)
     END DO

     CALL MPI_BSEND(A(N * nthreads),1,.....)
     CALL MPI_RECV(A(0),1,.....)

!$omp parallel do
     DO I = 1,N
        D(I) = A(I-1) + A(I)
     END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
     DO I=1,N * nthreads
        A(I) = B(I) + C(I)
     END DO

     CALL MPI_BSEND(A(N * nthreads),1,.....)
     CALL MPI_RECV(A(0),1,.....)

!$omp parallel do
     DO I = 1,N * nthreads
        D(I) = A(I-1) + A(I)
     END DO
```

Intra-node messages overlapped with inter-node

Example

```
!$omp parallel do
     DO I=1,N * nthreads
        A(I) = B(I) + C(I)
     END DO

     CALL MPI_BSEND(A(N * nthreads),1,.....)
     CALL MPI_RECV(A(0),1,.....)

!$omp parallel do
     DO I = 1,N * nthreads
        D(I) = A(I-1) + A(I)
     END DO
```

Intra-node messages overlapped with inter-node

Inter-thread communication occurs here

Example

```
!$omp parallel do
     DO I=1,N * nthreads

        A(I) = B(I) + C(I)

     END DO


     CALL MPI_BSEND(A(N * nthreads),1,.....)

     CALL MPI_RECV(A(0),1,.....)


!$omp parallel do
     DO I = 1,N * nthreads

        D(I) = A(I-1) + A(I)

     END DO
```

Intra-node messages overlapped with inter-node

Inter-thread communication occurs here

Example

```
!$omp parallel do
     DO I=1,N * nthreads

        A(I) = B(I) + C(I)

     END DO


     CALL MPI_BSEND(A(N * nthreads),1,.....)

     CALL MPI_RECV(A(0),1,.....)


!$omp parallel do
     DO I = 1,N * nthreads

        D(I) = A(I-1) + A(I)

     END DO
```

Implicit barrier added here

Intra-node messages overlapped with inter-node

Inter-thread communication occurs here

Example

```
!$omp parallel do
     DO I=1,N * nthreads
        A(I) = B(I) + C(I)
     END DO


     CALL MPI_BSEND(A(N * nthreads),1,.....)
     CALL MPI_RECV(A(0),1,.....)


!$omp parallel do
     DO I = 1,N * nthreads
        D(I) = A(I-1) + A(I)
     END DO
```

Implicit barrier added here

Intra-node messages overlapped with inter-node

Inter-thread communication occurs here

Funneled

- Advantages
  - relatively simple to write and maintain
  - cheaper ways to synchronise threads before and after message transfers
  - possible for other threads to compute while master is in an MPI call
- Disadvantages
  - less clear separation between outer (MPI) and inner (OpenMP) levels of parallelism
  - all communicated data still passes through the cache where the master thread is executing.
  - inter-process and inter-thread communication still do not overlap.

OpenMP Funneled with overlapping (1)

```
#pragma omp parallel
{
   … work
   #pragma omp barrier
   if (omp_get_thread_num() == 0)  {
      ierror=MPI_Send(…);
   }
   else {
      do some computation
   }
  #pragma omp barrier
   … work
}
```

Can't using worksharing here!

OpenMP Funneled with overlapping (2)

```
#pragma omp parallel num_threads(2)
{
if (omp_get_thread_num() == 0)  {
     ierror=MPI_Send(…);
  }
  else {
#pragma omp parallel
     {
         do some computation
     }
  }
}
```

Higher overheads and harder to synchronise between teams

Serialised

- Advantages
  - easier for other threads to compute while one is in an MPI call
  - can arrange for threads to communicate only their “own” data (i.e. the data they read and write).
- Disadvantages
  - getting harder to write/maintain
  - more, smaller messages are sent, incurring additional latency overheads
  - need to use tags or communicators to distinguish between messages from or to different threads in the same MPI process.

Distinguishing between threads

- By default, a call to MPI_Recv by any thread in an MPI process will match an incoming message from the sender.
- To distinguish between messages intended for different threads, we can use MPI tags
  - if tags are already in use for other purposes, this gets messy
- Alternatively, different threads can use different MPI communicators
  - OK for simple patterns, e.g. where thread N in one process only ever communicates with thread N in other processes
  - more complex patterns also get messy

Multiple

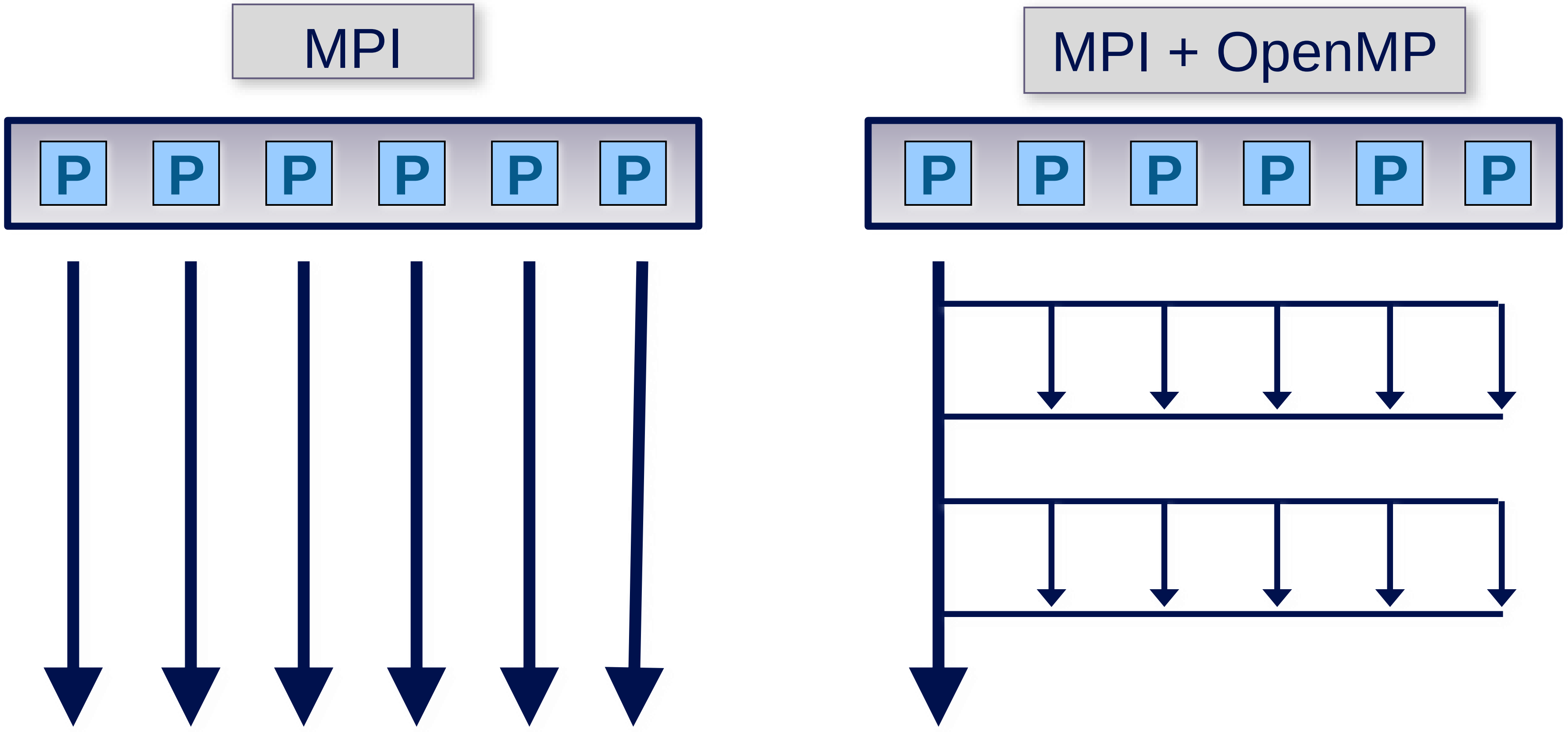
- Advantages
  - Messages from different threads can (in theory) overlap
    - many MPI implementations serialise them internally.
  - Natural for threads to communicate only their "own" data
  - Fewer concerns about synchronising threads (responsibility passed to the MPI library)
- Disdavantages
  - Hard to write/maintain
  - Not all MPI implementations support this – loss of portability
  - Most MPI implementations don't perform well like this
    - Thread safety implemented crudely using global locks.

End points

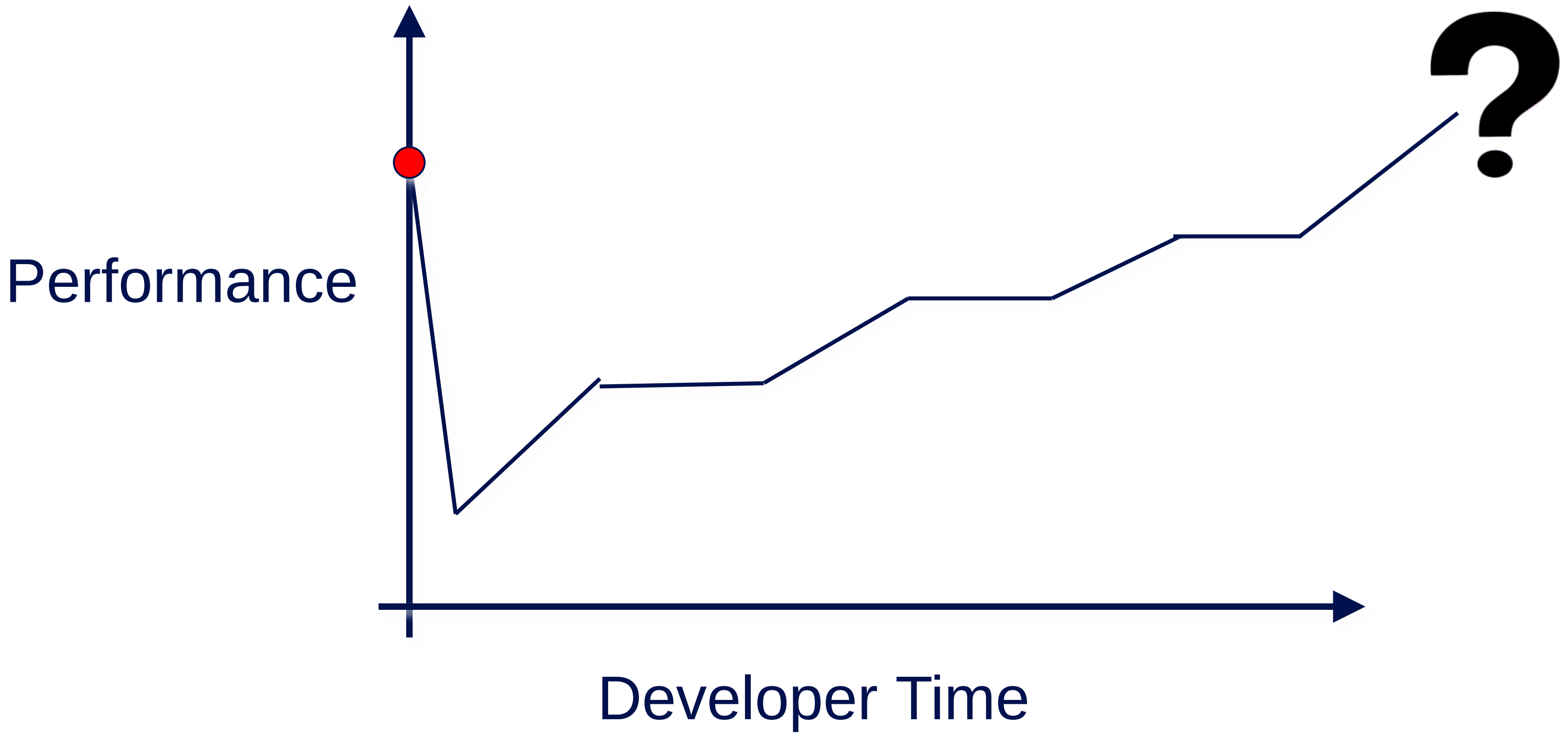
- A possible solution to permit more easier use and efficient implementations of Multiple is to extend MPI so that an MPI rank may have multiple source and destination identifiers (end points)
- e.g. if we want 4 threads per MPI process we could create an MPI communicator with 4 end points per rank
  - each thread can use a different end point
- Avoids need to use tags to identify threads
- Currently under discussion in MPI Forum
  - might appear in MPI 4.0?

Performance

- Conceptually easy to write
  - rather messy
  - hard to get good performance: cannot just concentrate on key kernels



Consequences



Summary

- Hybrid programming still a major current research topic
- Many see it as the key to exascale, however …
  - will require MPI_THREAD_MULTIPLE style to avoid synchronisation
  - ... and end points to make this usable?
- Achieving correctness is hard
  - have to consider race conditions on messages
- Achieving performance is hard
  - entire application must be threaded (efficiently!)
- Must optimise choice of
  - numbers of processes/threads
  - placement of processes/threads on NUMA architectures