Parallel Computing Notes
Topic: Notes on Hybrid MPI + OpenMP Programming

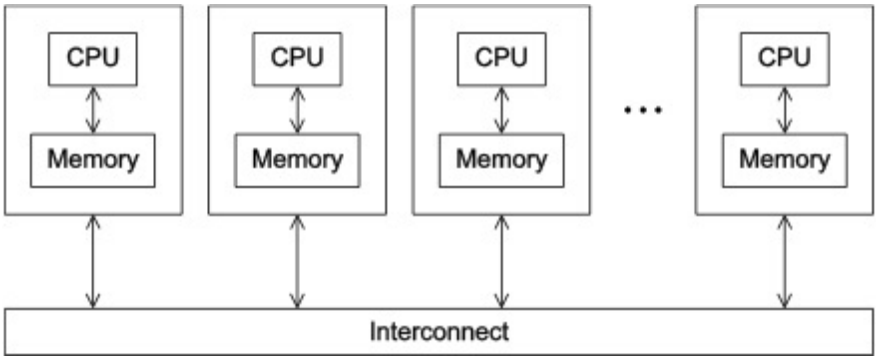
Mary Thomas

Department of Computer Science
Computational Science Research Center (CSRC)
San Diego State University (SDSU)

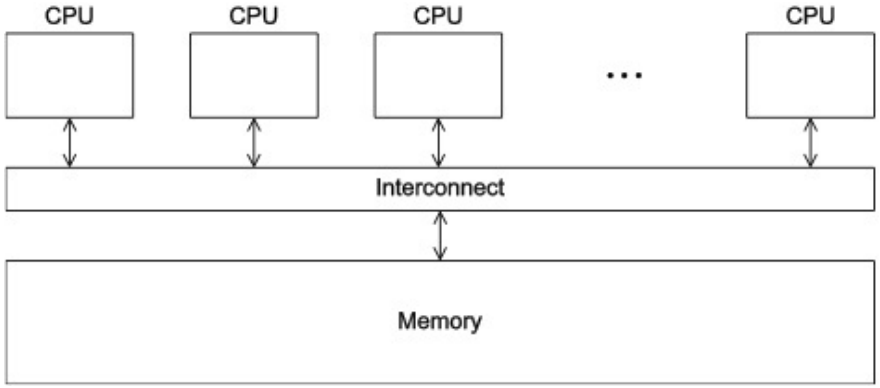
Last Update: 11/01/15

Table of Contents

Parallel Programming Models



MPI Distributed-memory system: collection of cores, connected with a network, each with its own memory.

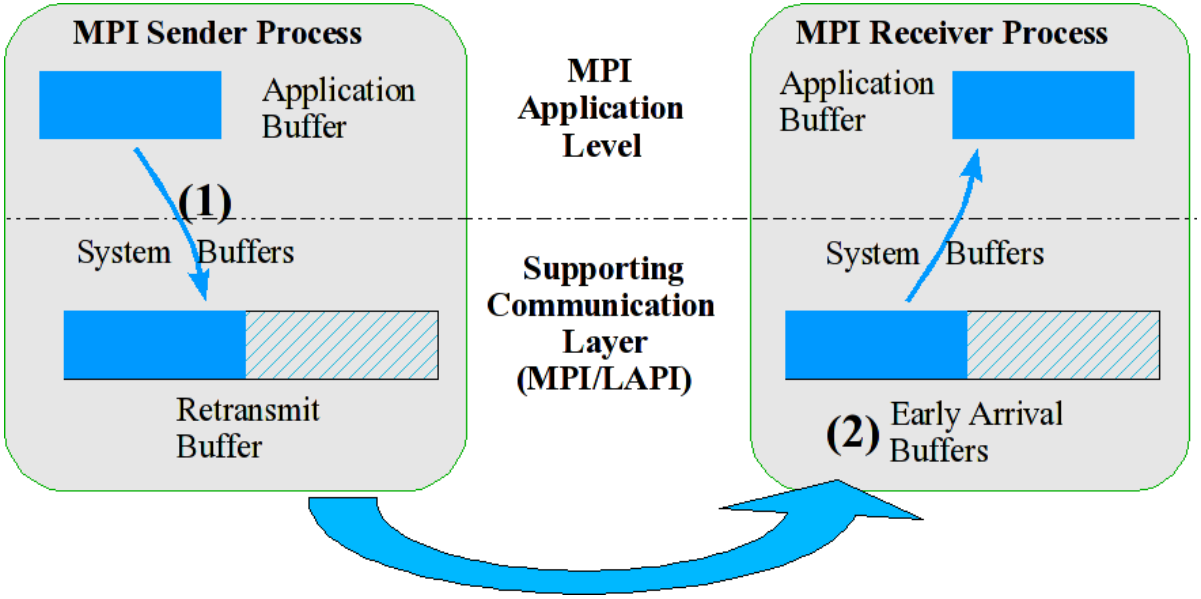


OpenMP Shared-memory system: collection of cores interconnected to a global memory.

MPI Overview

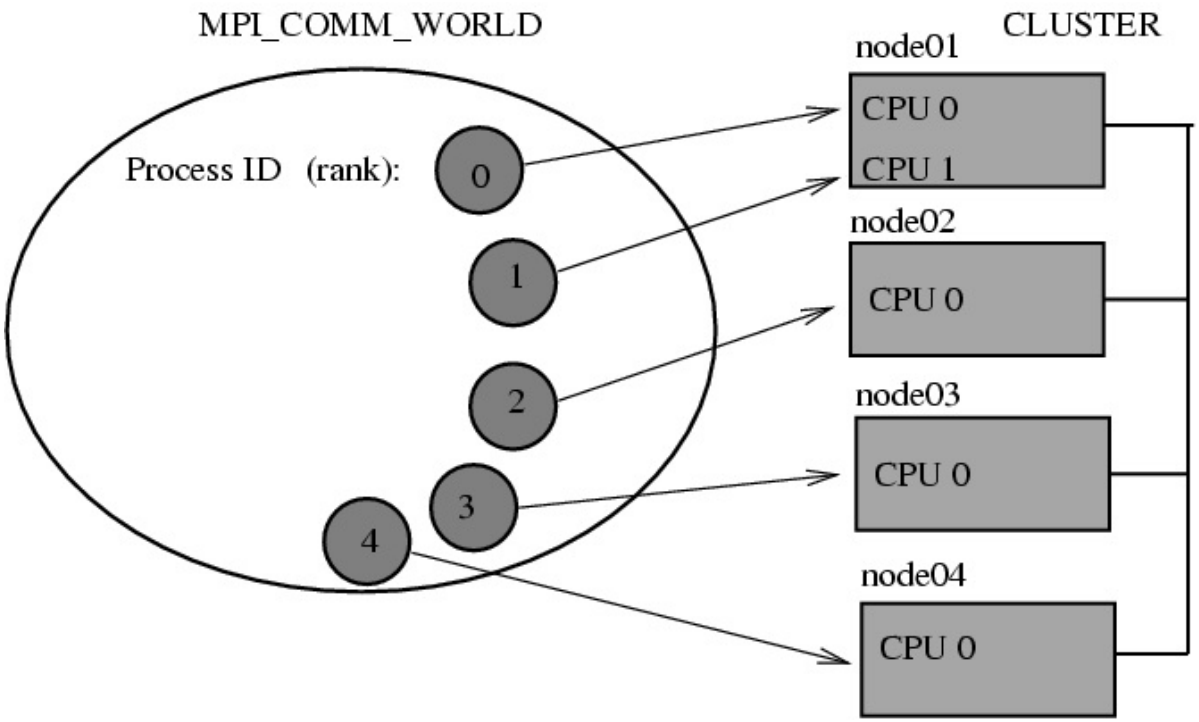
- For running codes on distributed memory systems.
- Data resides on other processes accessed through MPI calls.
- A framework for distributed-memory parallelism:
  - Multiple tasks run concurrently across separate nodes
  - Each task has its own private memory
  - Memory is shared by passing messages among nodes
  - Messaging requires a high-performance interconnect
- MPI is implemented as a library with wrappers for compiling: mpicc (C), mpic++ (C++), mpif90 (Fortran 90)
- The minimal set of routines that most parallel codes use:
  - MPI_INIT, MPI_FINALIZE
  - MPI_COMM_SIZE, MPI_COMM_RANK
  - MPI_SEND, MPI_RECV

MPI: Point-to-Point Communications



- **Point-to-Point:** uses **MPI_SEND** and **MPI_RECV**.
- Two PEs transfer data from one to the other *only*.
- Blocking: PE #1 posts a **SEND** operation
- Target (PE#2) process posts a **RECEIVE** for data being transferred.

Image Source: `http://sc.tamu.edu/systems/hydra/hardware.php`
Note: LAPI is the IBM Low-level Application Programming Interface



Communication on a multimode cluster with multiple cores (PEs). In this archtiectures, the cores do not share memory.

MPI Collective Communications

- All cores call the same MPI function at the same point in the program
- Arguments passed by each process to an MPI collective communication must be compatible.
- Collective communications matched by communicator and order called.
  Note: point-to-point communications are matched on the basis of tags and communicators.
- Common collective operations
  - MPI_Gather, MPI_Scatter, MPI_Reduce
  - MPI_Allgather, MPI_Allscatter, MPI_Allreduce
  - MPI_Bcast

MPI Virtual Topologies

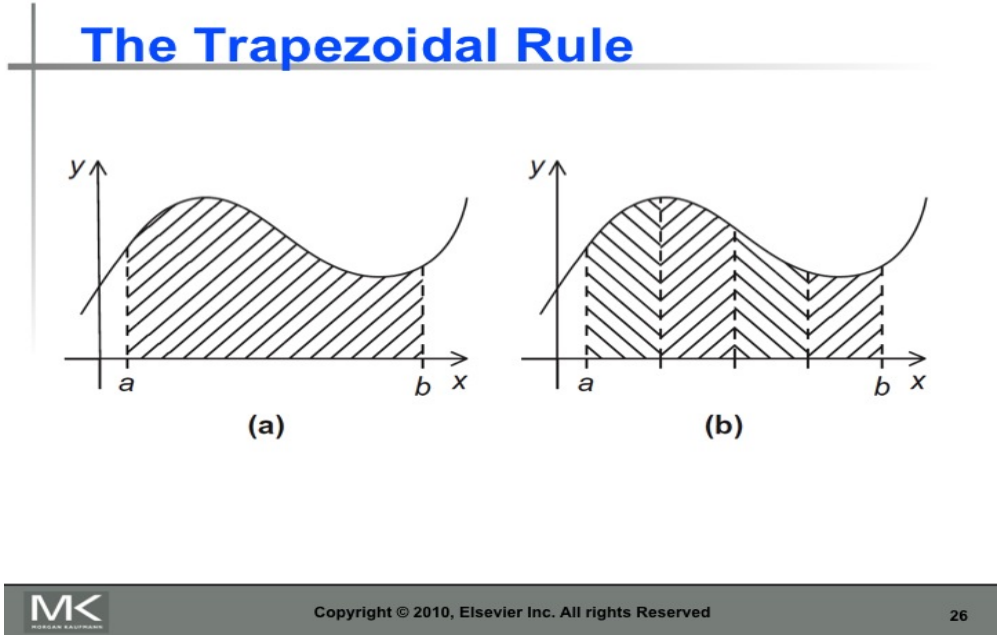
- MPI topologies are virtual - there may be no relation between the physical structure of the parallel machine and the process topology.
- Virtual topologies are built upon MPI communicators and groups.
- Must be "programmed" by the application developer.
- Two Types: Cartesian, Graphs
- Cartesian: 1D, 2D, 3D arrangements
- Convenient:
  - Useful for applications with specific communication patterns - patterns that match an MPI topology structure.
- Improved communication efficiency:
  - hardware architectures may impose penalties for communications between successively distant "nodes".
  - A particular implementation may optimize process mapping based upon the physical characteristics of a given parallel machine.
  - The mapping of processes into an MPI virtual topology is dependent upon the MPI implementation, and may be ignored.

MPI 3D Cartesian Mapping

- We have looked at MPI collective communication routines that optimize data distribution.
- Next, we need to look at ways to configure the processors to better match the geometry/approach needed to solve the scientific problem.
- Examples below use the following MPI Cartesian topologies:
  - MPI_Dims_create: Create N-Dimensional arrangement of PEs in the cartesian grid.
  - MPI_Cart_create:: Create N-Dimensional virtual topology/cartesian grid.
  - MPI_Cart_coords: Get local PE coordinates in the new cartesian grid
  - MPI_Cart_sub: Partitions a communicator into subgroups which form lower-dimensional cartesian subgrids.
  - MPI_Cart_shift: Used to find processor neighbors. Returns the shifted source and destination ranks, given a shift direction and amount.
- Today we will look more closely at how this works.

The Trapezoid Rule for Numerical Integration

Solve the Integral: $\int_a^b F(x)dx$



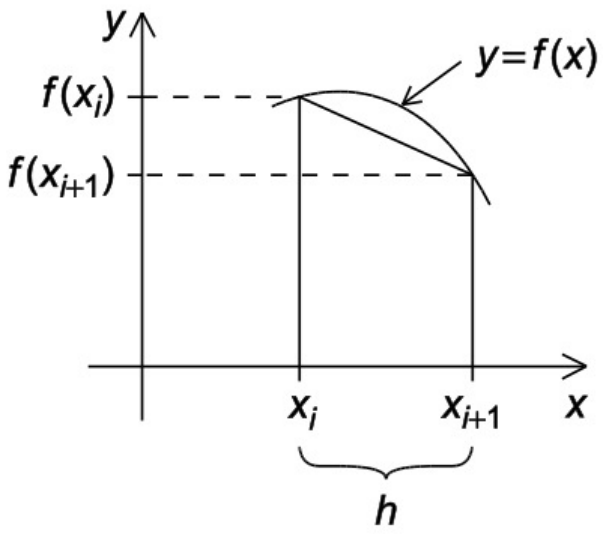
Where $F(x)$ can be any function of x: $f(x^2)$, $f(x^3)$
See Pacheco IPP (2011), Ch3.

Trapezoid Serial

Integral: $\int_a^b f(x)dx$

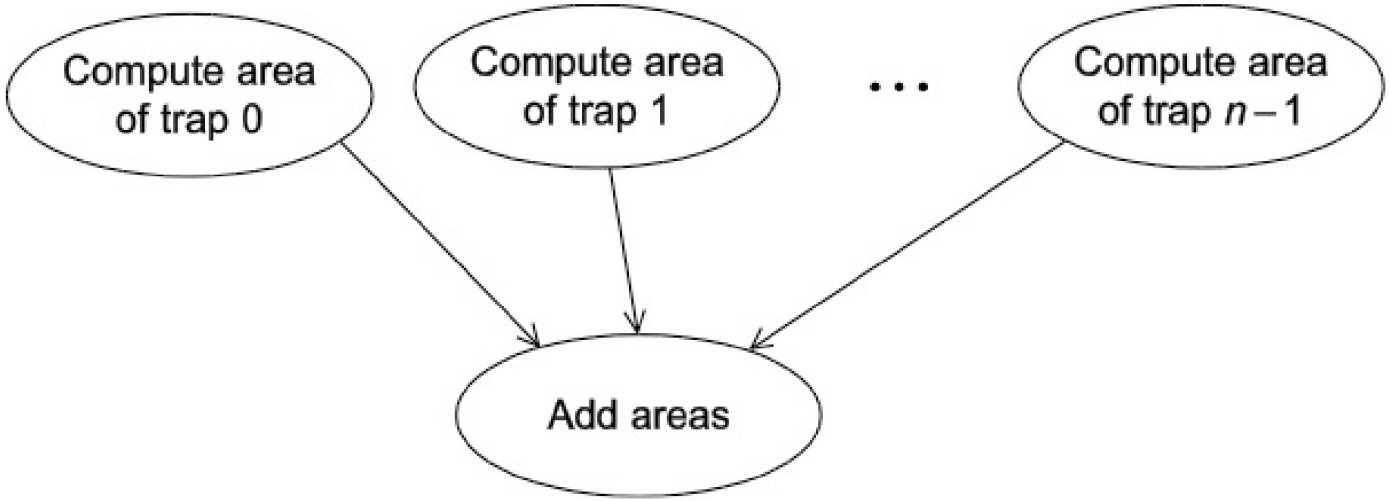
Area of 1 trapezoid: $= \frac{h}{2}\left[f(x_i) + f\left(x_{i+1}\right)\right]$

Base: $h = \frac{b-a}{n}$



Endpoints: $x_0 = a, \quad x_1 = a + h, \quad x_2 = a + 2h, \ldots, \quad x_{n-1} = a + (n-1)h, \quad x_c = b$

Sum of Areas: $Area = h\left[\frac{f(x_0)}{2} + f(x_{i+1}) + f(x_{i+1}) + \ldots + f(x_{n-1})\frac{f(x_n)}{2}\right]$



Two types of tasks:

- **Compute area of 1 trapezoid**
- **Compute area sums**

MPI Parallel Pseudocode

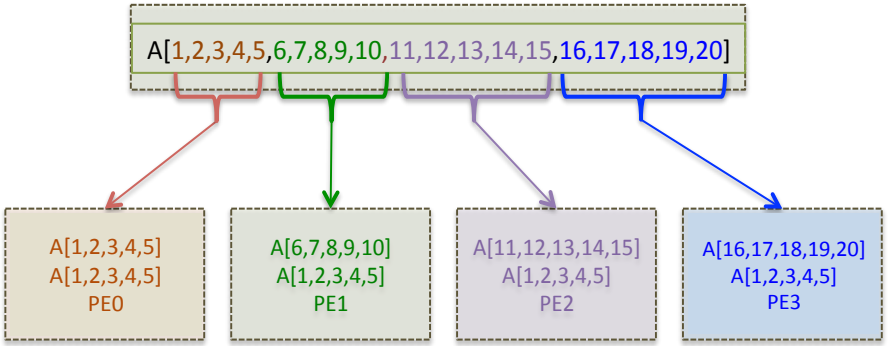
```
Get a,b,n;
h = b*a/n ;
local_n = n/comm_sz;
local_a = a + my_rank*local_n*h;
local_b = local_a + local_n*h;
local_integral = Trap(local_a, local_b, local_n, h);
if (my_rank != 0)
     Send local integral to process 0;
else      /* my_rank==0 */
     total_integral = local_integral;
     for (proc = 1; proc < comm_sz; proc++)
     {
          Receive local integral from proc;
          total integral += local integral;
     }
}

if (my_rank == 0)
print result;
```

```
/* File:     mpi_trap4.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <mpi.h>
#include <math.h>

double CalcPI(double left_endpt, double right_endpt, int trap_count,
   double base_len);

/* Function we're integrating */
double f_pi(double x);

int main(int argc, char** argv){
   int my_rank, comm_sz, n, local_n;
   double a, b, h, local_a, local_b;
   double local_int, total_int;

   struct timeval  tvalStart, tvalStop;
   struct timeval  tvalTmp, tvalElap;
   double Telap;

   gettimeofday (&tvalStart, NULL);

   /* Let the system do what it needs to start up MPI */
   MPI_Init(NULL, NULL);

   /* Get my process rank */
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   /* Find out how many processes are being used */
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
```

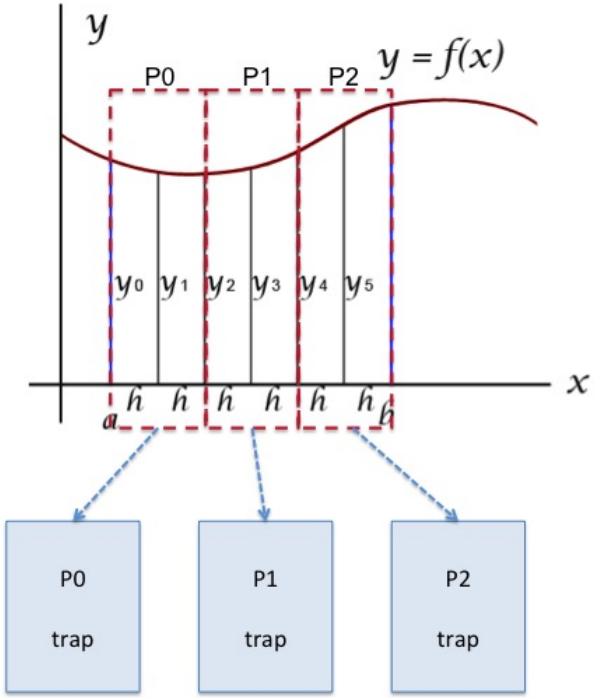
```
    if(argc != 2){
       if (my_rank == 0) {
          printf("Usage: mpi_trap_pi  n  \n");
       }
       MPI_Barrier(MPI_COMM_WORLD);
       MPI_Finalize();
    } else {
       n =  atoi (argv[1]);
       h = (b-a)/n;          /* h is the same for all processes */
       local_n = n/comm_sz;  /* So is the number of trapezoids  */

       /* Length of each process' interval of integration = local_n*h.
        * interval starts at: */
       local_a = a + my_rank*local_n*h;
       local_b = local_a + local_n*h;
       local_int = CalcPI(local_a, local_b, local_n, h);

       /* Add up the integrals calculated by each process */
       MPI_Reduce(&local_int, &total_int, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

       gettimeofday (&tvalStop, NULL);

       Telap= (double)( (tvalStop.tv_sec - tvalStart.tv_sec)*1.0E6
            +tvalStop.tv_usec - tvalStart.tv_usec ) / 1.0E6;

       /* Print the result */
       if (my_rank == 0) {
           printf("With n = %d trapezoids, our estimate\n", n);
           printf("   Estimated value of  pi = %.14f\n",total_int);
           printf("   Reference value of  pi = %.14f\n", 4.0*atan(1.0));
          printf("   Esitmate Error of   pi = %.15e\n", fabs(total_int - 4.0*atan(1.0)) );
          printf("   Telapsed in seconds: %e seconds\n", Telap);
       }
    }
    /* Shut down MPI */
    MPI_Finalize();
    return 0;
 } /*  main  */
```

```
double CalcPI(
      double left_endpt  /* in */,
      double right_endpt /* in */,
      int    trap_count  /* in */,
      double base_len    /* in */) {
   double estimate, x;
   int i;

   estimate = (f_pi(left_endpt) + f_pi(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
      x = left_endpt + i*base_len;
      //estimate += 4.0/(1+x*x);
      estimate += f_pi(x);
   }
   estimate = estimate*base_len;

   return estimate;
} /*  CalcPI  */

/*-----------------------------------------------------------------
 * Function:    f_pi
 * Purpose:     Compute value of function to be integrated
 * Input args:  x
 */
double f_pi(double x /* in */) {
   return 4.0/(1+x*x);
} /* f_pi */
```

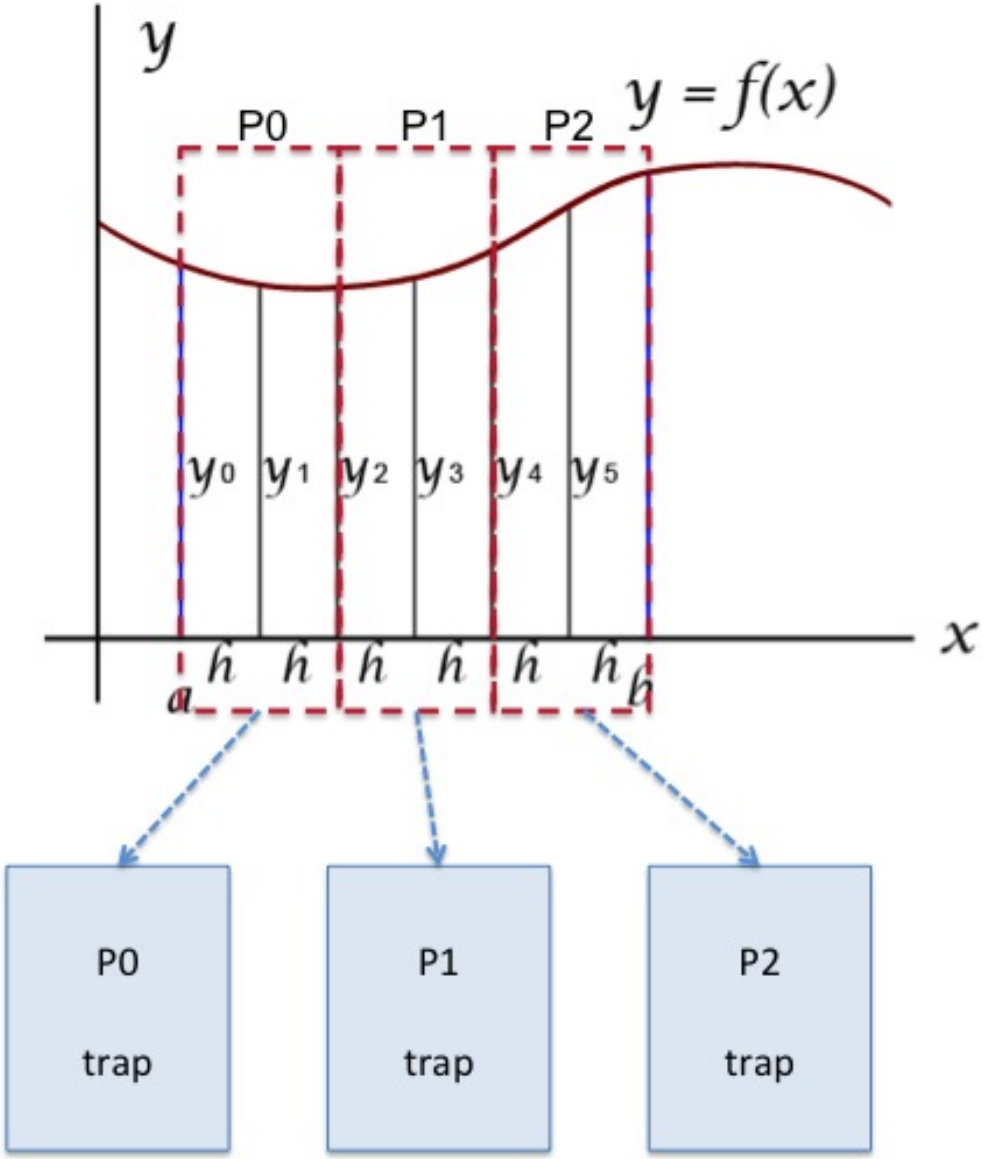
MPI Data Distribution for Trap Function



1D Vector Data Distribution



- Parallel *trap* distributes $[x_i]$ data points to different processors.
- Each processor runs the same code but solves for different $[x_i]$ data
- This is a *1-D* data decomposition
- This is a *1-D* (virtual) processor arrangement

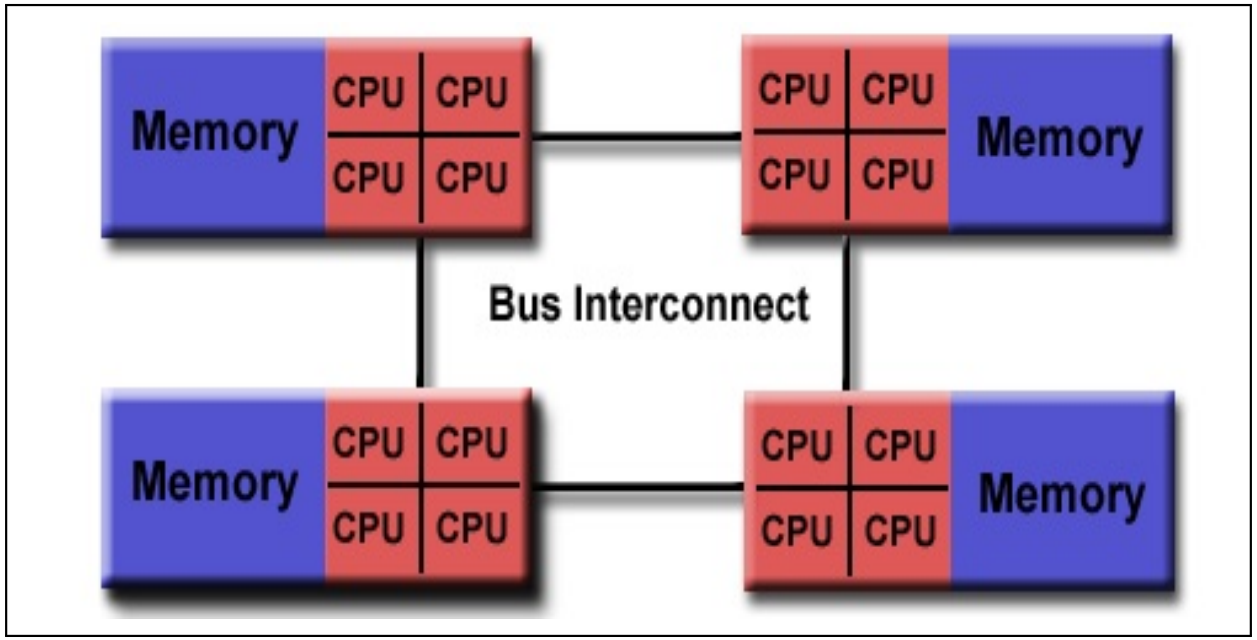

y
y = f(x)
P0
P1
P2
y0
y1
y2
y3
y4
y5
x
a
h
b
P0
trap
P1
trap
P2
trap

OpenMP Overview

- OpenMP = Open MultiProcessing
- API that supports multi-platform shared memory multiprocessing programming.
- Designed for systems in which each thread or process can potentially have access to all available memory.
- System is viewed as a collection of cores or CPUs, all of which have access to main memory
- Applications built using hybrid model of parallel programming:
  - Runs on a computer cluster using both OpenMP and Message Passing Interface (MPI)
  - OR through the use of OpenMP extensions for non-shared memory systems.
- See:
  - `http://openmp.org/`
  - `http://en.wikipedia.org/wiki/OpenMP`

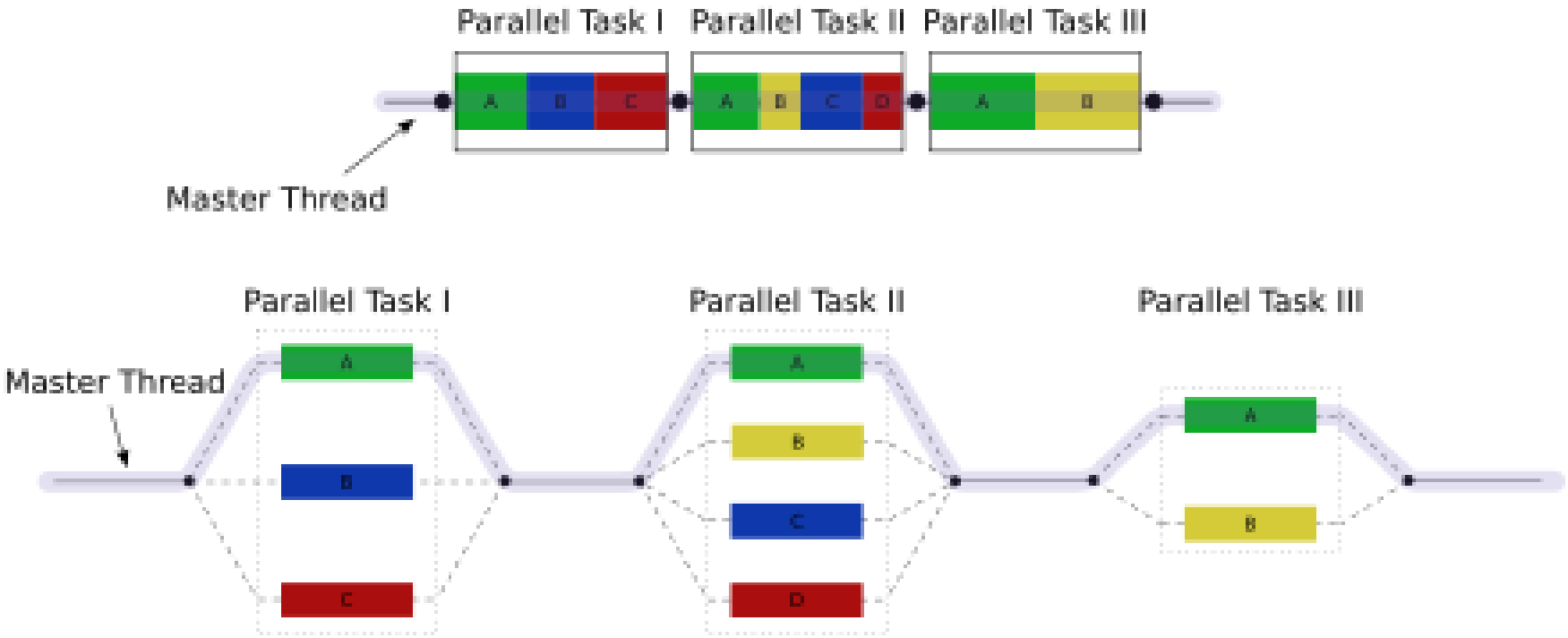
What is OpenMP?

- OpenMP grew out of the need to standardize different vendor specific directives related to parallelism.
- Pthreads not scaleable to large systems and does not support incremental parallelism very well.
- Correlates with evolution of hybrid architectures: shared memory and multi PE architectures being developed in early '90s.
- Structured around parallel loops and was meant to handle dense numerical applications.



Source: `https://computing.llnl.gov/tutorials/openMP`

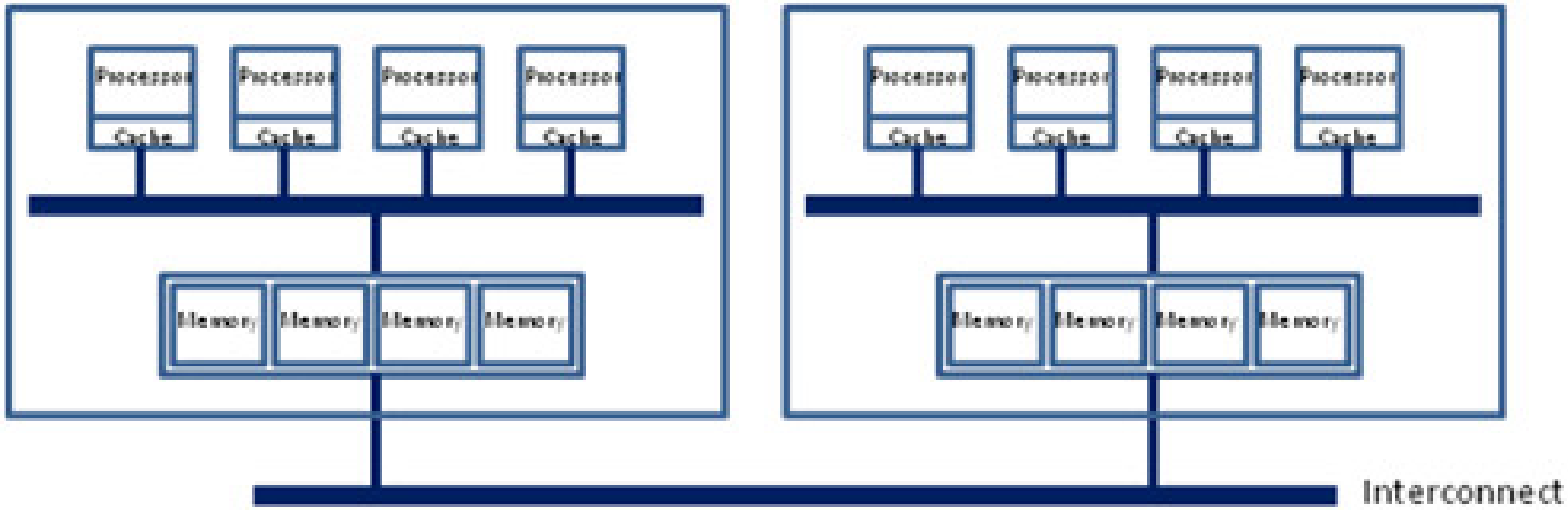
OpenMP is an implementation of *multithreading*



Source: `http://en.wikipedia.org/wiki/OpenMP`

- Method of parallelizing where a master thread forks a specified number of slave threads
- Tasks are divided among them.
- Threads run concurrently.

Non Uniform Memory Access (NUMA)



- Hierarchical Scheme: processors are grouped by physical location
- located on separate multi-core (PE) CPU packages or nodes.
- Processors (PEs) within a node share access to memory modules via UMA shared memory architecture.
- PE's may also access memory from the remote node using a shared interconnect

Source: `https://software.intel.com/en-us/articles/optimizing-applications-for-numa`

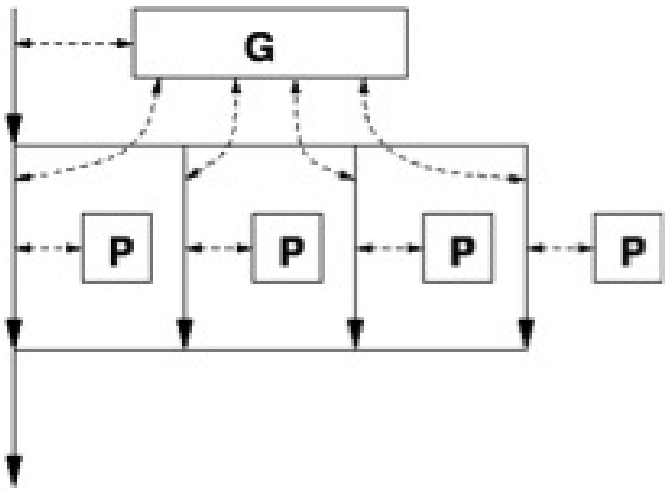
OpenMP: General Code Structure

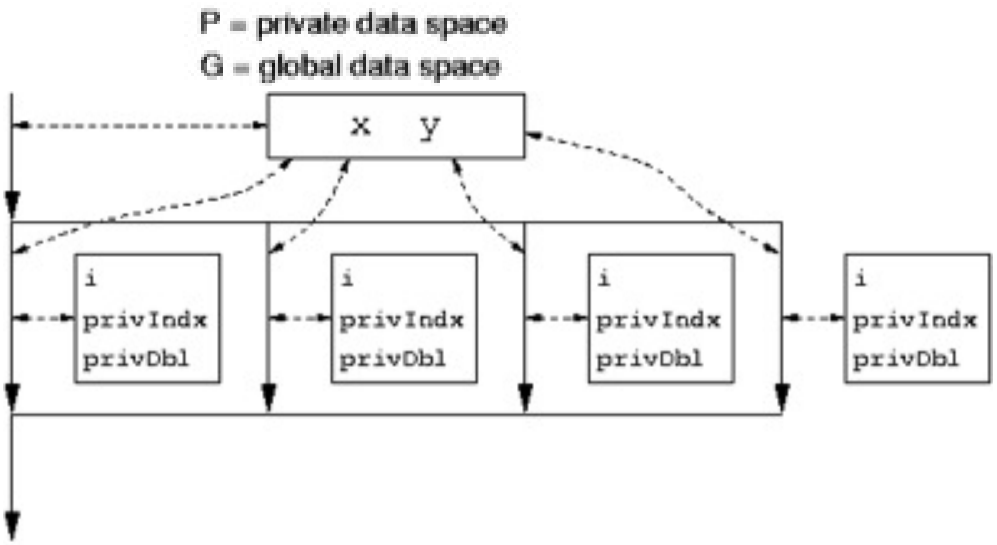
```
 #include <omp.h>
main () {
   int var1, var2, var3;
   Serial code
   . . .
  /* Beginning of parallel section.
   Fork a team of threads. Specify variable scoping*/
   #pragma omp parallel private(var1, var2) shared(var3)
   {
      /* Parallel section executed by all threads */
     . . .
     /* All threads join master thread and disband*/
    }
    Resume serial code
    . . .
}
```

OpenMP: Data Model

- Private and shared variables
- Global data space: accessed by all parallel threads.
- Private space: only be accessed by the thread.
- Parallel for loop index private by default.

```
#pragma omp parallel for private(
     privIndx, privDbl )
  for ( i = 0; i < arraySize; i++){
     for(privdx=0; privdx <16;privdx++){
        privDbl=( (double)privdx)/16;
      y[i]=sin(exp(cos( -exp(sin(x[i])))))
                    + cos( privDbl );
   }    }
```





execution context for "arrayUpdate_II"

OpenMP: Hello World

```
/* File:     omp_hello.c
 * Purpose:  A parallel hello, world program that uses OpenMP
 * Compile:  gcc -g -Wall -fopenmp -o omp_hello omp_hello.c
 * Run:      ./omp_hello <number of threads>
 * Input:    none
 * Output:   A message from each thread
 * IPP:      Section 5.1 (pp. 211 and ff.)
 */
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void);  /* Thread function */

/*-----------------------------------------------------------------*/
int main(int argc, char* argv[]) {
   int thread_count = strtol(argv[1], NULL, 10);

#  pragma omp parallel num_threads(thread_count)
   Hello();

   return 0;
}  /* main */

/*-----------------------------------------------------------------
 * Function:    Hello
 * Purpose:     Thread function that prints message
 */
void Hello(void) {
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   printf("Hello from thread %d of %d\n", my_rank, thread_count);

}  /* Hello */
```

Compiling and Running OpenMP Hello World

```
[mthomas]%
[mthomas@tuckoo]$ mpicc -g -Wall -fopenmp -o omp_hello omp_hello.c
```

```
[mthomas@tuckoo ch5]$ ./omp_hello 10
Hello from thread 6 of 10
Hello from thread 4 of 10
Hello from thread 5 of 10
Hello from thread 0 of 10
Hello from thread 1 of 10
Hello from thread 7 of 10
Hello from thread 2 of 10
Hello from thread 3 of 10
Hello from thread 9 of 10
Hello from thread 8 of 10
```

OpenMP Directive: #pragma

```
#  pragma omp parallel num_threads(thread_count)
  Hello();
```

- #pragma is first OpenMP directive.
- Scope of a directive is one block of statements {...}
- OpenMP determines # threads to create, synchronize, destroy
- Start threads running thread function Hello.
- *num_threads(thread_count)* is an OpenMP clause
- Similar (but less work) to the Pthread command:

```
pthread_create(&thread_handles[i], NULL, Thread_work, (void*) i);
```

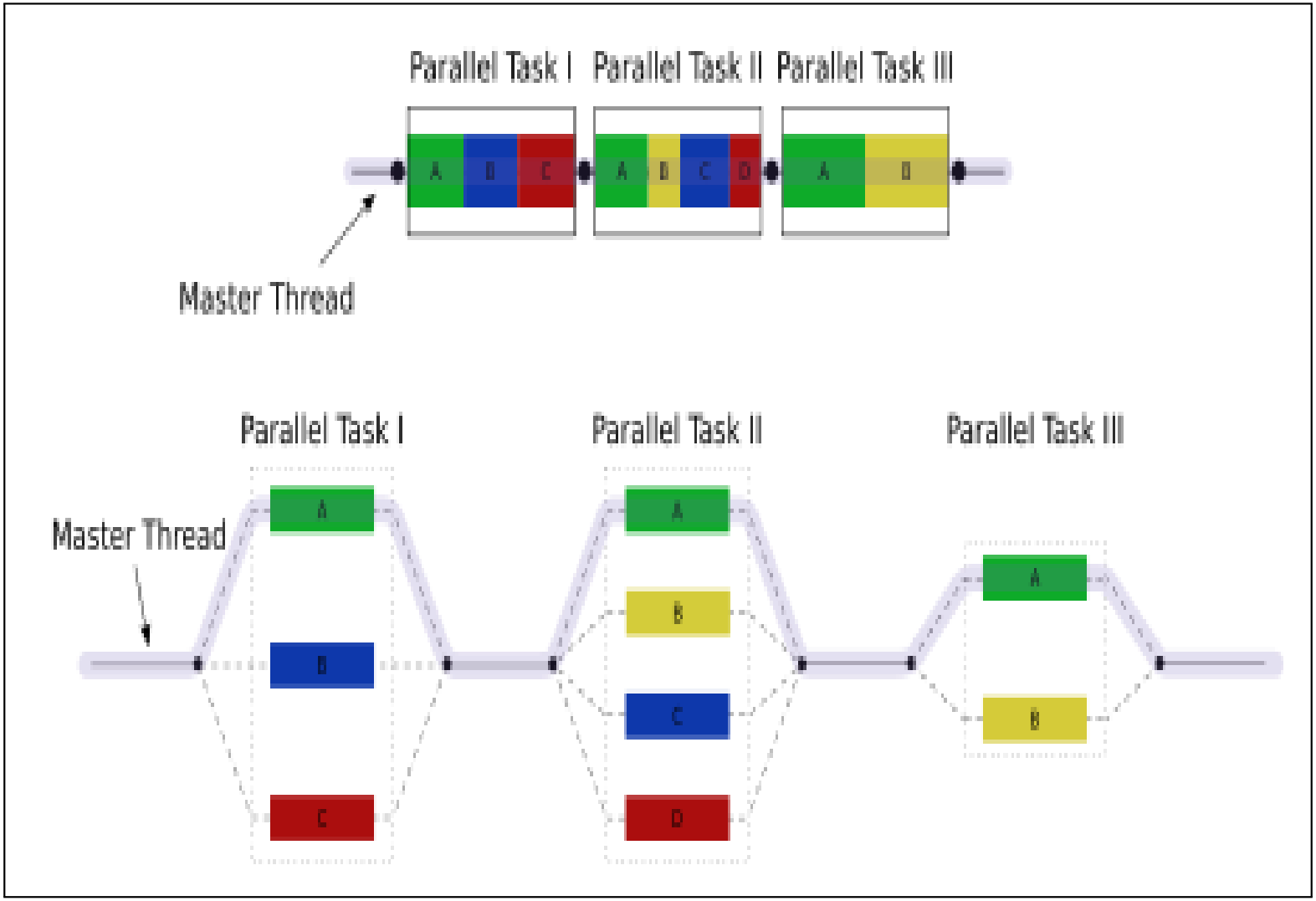
- Special preprocessor instructions.
- Typically added to a system to allow behaviors that arent part of the basic C specification.
- Portable: compilers that don't support the pragmas ignore them.

OpenMP: Parallel Region Construct

- Defines a block of code to be executed by the threads:

```
#  pragma omp parallel num_threads(thread_count)
        {
                ...
            } (implied barrier)
```

- Example clauses:
  - if (expression): only in parallel if expression evaluates to true
  - private(list): everything private and local (no relation with variables outside the block).
  - shared(list): data accessed by all threads
  - default (none — shared)
  - reduction (operator: list)
  - firstprivate(list), lastprivate(list)


Parallel Task I
Parallel Task II
Parallel Task III
A
B
C
D
Master Thread
Parallel Task I
Parallel Task II
Parallel Task III
Master Thread
A
B
C
D

Trapezoid Algorithm - Serial

```
/*  Input: a ,b, n  */
h = (b-a)/n ;
approx - (F(a) + F(b))/2.0
for (i=0; i<= n-1; i++) {
   x_i = a + i*H;
   approx += f(x_i);
}
approx = h* approx
```

OMP Trapezoid

```
/* File:    omp_trap1.c,
 * Pacheco IPP:  Section 5.2.1 (pp. 216 and ff.)     */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

void Usage(char* prog_name);
double f(double x);    /* Function we're integrating */
void Trap(double a, double b, int n, double* global_result_p);
double newglobres;

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;
   newglobres=0;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);
#  pragma omp parallel num_threads(thread_count)
   Trap(a, b, n, &global_result);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e, new%.14e\n",
      a, b, global_result, newglobres);
   return 0;
}  /* main */
```

OMP Trapezoid

```
/*-----------------------------------------------------------------
 * Function:    Usage
 * Purpose:     Print command line for function and terminate
 * In arg:      prog_name
 */
void Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "   number of trapezoids must be evenly divisible by\n");
   fprintf(stderr, "   number of threads\n");
   exit(0);
}  /* Usage */

/*----------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input arg:   x
 * Return val:  f(x)
 */
double f(double x) {
   double return_val;

   return_val = x*x;
   return return_val;
}  /* f */
```

OMP Trapezoid

```
/*-----------------------------------------------------------------
 * Function:    Trap
 * Purpose:     Use trapezoidal rule to estimate definite integral
 * Input args:
 *    a: left endpoint
 *    b: right endpoint
 *    n: number of trapezoids
 * Output arg:
 *    integral:  estimate of integral from a to b of f(x)
 */
void Trap(double a, double b, int n, double* global_result_p) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

   newglobres += my_result;

#  pragma omp critical
   {
   *global_result_p += my_result;
   }
}  /* Trap */
```

OpenMP: Reduction Clause

We need this more complex version to add each thread's local calculation to get *global_result*.

```
void Trap(double a, double b, int n, double* global_result_p);
```

Although we'd prefer this.

```
double Trap(double a, double b, int n);
```

```
global_result = Trap(a, b, n);
```

If we use this, there's no critical section!

```
double Local_trap(double a, double b, int n);
```

If we fix it like this...

```
   global_result = 0.0;
#  pragma omp parallel num_threads(thread_count)
   {
#     pragma omp critical
      global_result += Local_trap(double a, double b, int n);
   }
```

... we force the threads to execute sequentially.

 30

Local_Trap does not have reference to the global variable global_result

We can avoid this problem by declaring a private variable inside the parallel block and moving the critical section after the function call.

```
    global_result = 0.0;
#   pragma omp parallel num_threads(thread_count)
    {
       double my_result = 0.0;  /* private */

       my_result += Local_trap(double a, double b, int n);
#      pragma omp critical
       global_result += my_result;
    }
```


Notes: the call to Local_Trap is inside the parallel block, but outside critical section; my_result is private to each thread

Reduction operators

- A reduction operator is a binary operation (such as addition or multiplication).
- A reduction is a computation that repeatedly applies the same reduction operator to a sequence of operands in order to get a single result.
- All of the intermediate results of the operation should be stored in the same variable: the reduction variable.

OMP Trapezoid

```
/* File:    omp_trap2b.c
 * Purpose: Estimate definite integral (or area under curve) using trapezoidal
 *          rule.  This version uses a reduction clause.
 * Pacheco IPP:  Section 5.4 (pp. 223 and ff.)
 */
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>

void Usage(char* prog_name);
double f(double x);    /* Function we're integrating */
double Local_trap(double a, double b, int n);

int main(int argc, char* argv[]) {
   double  global_result = 0.0;  /* Store result in global_result */
   double  a, b;                 /* Left and right endpoints      */
   int     n;                    /* Total number of trapezoids    */
   int     thread_count;

   if (argc != 2) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   printf("Enter a, b, and n\n");
   scanf("%lf %lf %d", &a, &b, &n);
   if (n % thread_count != 0) Usage(argv[0]);

#  pragma omp parallel num_threads(thread_count) \
      reduction(+: global_result)
   global_result += Local_trap(a, b, n);

   printf("With n = %d trapezoids, our estimate\n", n);
   printf("of the integral from %f to %f = %.14e\n",
      a, b, global_result);
   return 0;
}  /* main */
```

OMP Trapezoid

```
/*-----------------------------------------------------------------
 * Function:    Usage
 * Purpose:     Print command line for function and terminate
 * In arg:      prog_name
 */
void Usage(char* prog_name) {

   fprintf(stderr, "usage: %s <number of threads>\n", prog_name);
   fprintf(stderr, "   number of trapezoids must be evenly divisible by\n");
   fprintf(stderr, "   number of threads\n");
   exit(0);
}  /* Usage */

/*----------------------------------------------------------------
 * Function:    f
 * Purpose:     Compute value of function to be integrated
 * Input arg:   x
 * Return val:  f(x)
 */
double f(double x) {
   double return_val;

   return_val = x*x;
   return return_val;
}  /* f */
```

OMP Trapezoid

```
/*-----------------------------------------------------------------
 * Function:    Local_trap
 * Purpose:     Use trapezoidal rule to estimate part of a definite
 *              integral
 * Input args:
 *    a: left endpoint
 *    b: right endpoint
 *    n: number of trapezoids
 * Return val:  estimate of integral from local_a to local_b
 *
 * Note:        return value should be added in to an OpenMP
 *              reduction variable to get estimate of entire
 *              integral
 */
double Local_trap(double a, double b, int n) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();

   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;
   my_result = (f(local_a) + f(local_b))/2.0;
   for (i = 1; i <= local_n-1; i++) {
     x = local_a + i*h;
     my_result += f(x);
   }
   my_result = my_result*h;

   return my_result;
}  /* Trap */
```

OMP Trapezoid

```
#include <stdio.h>
#include "mpi.h"
#include <omp.h>

int main(int argc, char *argv[]) {
  int numprocs, rank, namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  int iam = 0, np = 1;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Get_processor_name(processor_name, &namelen);

  /* set number of threads from command line argument */
  int thread_count = strtol(argv[1], NULL, 10);
#  pragma omp parallel num_threads(thread_count)

  /* set number of threads using environment variables
   *  export OMP_NUM_THREADS=4
   */
  // #pragma omp parallel default(shared) private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d from process %d out of %d on %s\n",
           iam, np, rank, numprocs, processor_name);
  }

  MPI_Finalize();
}
```

```
[mthomas@tuckoo mpi.omp]$ ./mpi_omp_hello 4
Hello from thread 0 out of 4 from process 0 out of 1 on tuckoo
Hello from thread 2 out of 4 from process 0 out of 1 on tuckoo
Hello from thread 2 out of 4 from process 0 out of 1 on tuckoo
Hello from thread 1 out of 4 from process 0 out of 1 on tuckoo

[mthomas@tuckoo mpi.omp]$ ./mpi_omp_hello 8
Hello from thread 0 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 6 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 4 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 7 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 3 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 5 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 1 out of 8 from process 0 out of 1 on tuckoo
Hello from thread 2 out of 8 from process 0 out of 1 on tuckoo
```

```
/* File:     mpi_omp_pi.c */
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
#include <math.h>
#include <mpi.h>
#include <omp.h>

void Usage(char* prog_name);

double OMP_CalcPI(double a, double b, int n);
double CalcPI(double left_endpt, double right_endpt, int trap_count,
   double base_len);

/* Function we're integrating */
double f_pi(double x);

int main(int argc, char** argv){
   int my_rank, comm_sz, ierr;
   int n, local_n;
   int mpi_n, mpi_loc_n;
   double mpi_a,mpi_b,mpi_h;
   double mpi_loc_a,mpi_loc_b;
   double mpi_global_pi;

   double omp_global_pi;
   int thread_count;

   struct timeval  tvalStart, tvalStop;
   struct timeval  tvalTmp, tvalElap;
   double Telap;

   gettimeofday (&tvalStart, NULL);

   /* Let the system do what it needs to start up MPI */
   MPI_Init(NULL, NULL);
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
   MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
```

```
  if (argc != 3) {
   if (my_rank == 0)  Usage(argv[0]);
      MPI_Abort(MPI_COMM_WORLD,ierr);
   }

   if (argc != 3) Usage(argv[0]);
   thread_count = strtol(argv[1], NULL, 10);
   n =  atoi (argv[2]);
   local_n = n/comm_sz;  /* So is the number of trapezoids  */

   /* separate mpi and omp vars */
   mpi_n       = atoi (argv[2]);
   mpi_loc_n = mpi_n/comm_sz;  /* number of trapezoids on node  */

   /* Length of each process' interval of integration = local_n*h.  */
   /*  interval starts/ends at: */
   mpi_a = 0.0;
   mpi_b = 1.0;
   mpi_h = (mpi_b - mpi_a)/mpi_n;
   mpi_loc_a = mpi_a + my_rank*mpi_loc_n*mpi_h;
   mpi_loc_b = mpi_loc_a + mpi_loc_n*mpi_h;
   printf("P[%d]: mpi_n=%d, mpi_h,a,b=[%.8f,%.8f,%.8f],  mpi_loc_n,a,b=[%d,%.8f,%.8f]\n",
              my_rank,mpi_n, mpi_h,mpi_a,mpi_b, mpi_loc_n,mpi_loc_a,mpi_loc_b);

   omp_global_pi=0.0;
#  pragma omp parallel num_threads(thread_count) \
      reduction(+: omp_global_pi)
   omp_global_pi += OMP_CalcPI(mpi_loc_a, mpi_loc_b, mpi_loc_n);

   /* Add up the integrals calculated by each process */
   MPI_Reduce(&omp_global_pi, &mpi_global_pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

   gettimeofday (&tvalStop, NULL);

   Telap= (double)( (tvalStop.tv_sec - tvalStart.tv_sec)*1.0E6
           +tvalStop.tv_usec - tvalStart.tv_usec ) / 1.0E6;
```

```
  /* Print the result */
   if (my_rank == 0) {
      printf("With n = %d trapezoids, our estimate\n", n);
      printf("   Estimated value of  pi = %.14f\n",mpi_global_pi);
      printf("   Reference value of  pi = %.14f\n", 4.0*atan(1.0));
      printf("   Esitmate Error of   pi = %.15e\n", fabs(mpi_global_pi - 4.0*atan(1.0)) );
      printf("   Telapsed in seconds: %e seconds\n", Telap);
   }

   /* Shut down MPI */
   MPI_Finalize();

   return 0;
} /*  main  */

/*-----------------------------------------------------------------
 * Function:  Usage
 * Purpose:   Print a message explaining how to run the program
 * In arg:    prog_name
 */
void Usage(char* prog_name) {
   fprintf(stderr, "usage: %s <thread_count> <n>\n", prog_name);  /* Change */
   fprintf(stderr, "   thread_count is the number of threads >= 1\n");  /* Change */
   fprintf(stderr, "   n is the number of terms and should be >= 1\n");
   fprintf(stderr, "   n '%' thread_count == 0 \n");
   exit(0);
}  /* Usage */
```

```
/* Function:    OMP_CalcPI  */
double OMP_CalcPI(double a, double b, int n) {
   double  h, x, my_result;
   double  local_a, local_b;
   int  i, local_n;
   int my_rank = omp_get_thread_num();
   int thread_count = omp_get_num_threads();
   h = (b-a)/n;
   local_n = n/thread_count;
   local_a = a + my_rank*local_n*h;
   local_b = local_a + local_n*h;

   my_result = CalcPI(local_a, local_b, local_n, h);

   return my_result;
}  /* OMP_CalcPI */

/*-------------------------------------------------------------------
double CalcPI( double left_endpt  /* in */,
          double right_endpt /* in */,
          int    trap_count  /* in */,
          double base_len    /* in */) {
     double estimate, x;
     int i;
    estimate = (f_pi(left_endpt) + f_pi(right_endpt))/2.0;
   for (i = 1; i <= trap_count-1; i++) {
        x = left_endpt + i*base_len;
       estimate += f_pi(x);
   }
   estimate = estimate*base_len;
   return estimate;
} /*  CalcPI  */

/*-------------------------------------------------------------------
 * Function:    f_pi   */
double f_pi(double x /* in */) {
   return 4.0/(1+x*x);
} /* f_pi */
```

```
[mthomas@tuckoo:~/pardev/matmul/omp/llnl] cat omp_mm.c
/******************************************************************************
* FILE: omp_mm.c
* DESCRIPTION:   OpenMp Example - Matrix Multiply - C Version
*   Demonstrates a matrix multiply using OpenMP. Threads share row iterations
*   according to a predefined chunk size.
* https://computing.llnl.gov/tutorials/openMP/samples/C/omp_mm.c
* AUTHOR: Blaise Barney,    LAST REVISED: 06/28/05
******************************************************************************/
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

//#define NRA 62                 /* number of rows in matrix A */
//#define NCA 15                 /* number of columns in matrix A */
//#define NCB 7                  /* number of columns in matrix B */
#define NRA 5                 /* number of rows in matrix A */
#define NCA 5                 /* number of columns in matrix A */
#define NCB 5                  /* number of columns in matrix B */

int main (int argc, char *argv[])  {
int        tid, nthreads, i, j, k, chunk;
double        a[NRA][NCA],           /* matrix A to be multiplied */
        b[NCA][NCB],           /* matrix B to be multiplied */
        c[NRA][NCB];           /* result matrix C */
chunk = 10;                    /* set loop iteration chunk size */
```

```
/*** Spawn a parallel region explicitly scoping all variables ***/
#pragma omp parallel shared(a,b,c,nthreads,chunk) private(tid,i,j,k) {
  tid = omp_get_thread_num();
  if (tid == 0)  {
    nthreads = omp_get_num_threads();
    printf("Starting matrix multiple example with %d threads\n",nthreads);
    printf("Initializing matrices...\n");        }
  /*** Initialize matrices ***/
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCA; j++)
      a[i][j]= i+j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NCA; i++)
    for (j=0; j<NCB; j++)
      b[i][j]= i*j;
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    for (j=0; j<NCB; j++)
      c[i][j]= 0;

  /*** Do matrix multiply sharing iterations on outer loop ***/
  /*** Display who does which iterations for demonstration purposes ***/
  printf("Thread %d starting matrix multiply...\n",tid);
  #pragma omp for schedule (static, chunk)
  for (i=0; i<NRA; i++)
    {
    printf("Thread=%d did row=%d\n",tid,i);
    for(j=0; j<NCB; j++)
      for (k=0; k<NCA; k++)
        c[i][j] += a[i][k] * b[k][j];
    }
  }   /*** End of parallel region ***/
```

```
/*** Print results ***/
printf("*******************************************************\n");
printf("Result Matrix:\n");
for (i=0; i<NRA; i++)
  {
  for (j=0; j<NCB; j++)
    printf("%6.2f   ", c[i][j]);
  printf("\n");
  }
printf("*******************************************************\n");
printf ("Done.\n");
return;
}
```