OpenMP Tutorial
Part 1: The Core Elements of OpenMP

Tim Mattson

Intel Corporation

Computational Software Laboratory

Rudolf Eigenmann

Purdue University

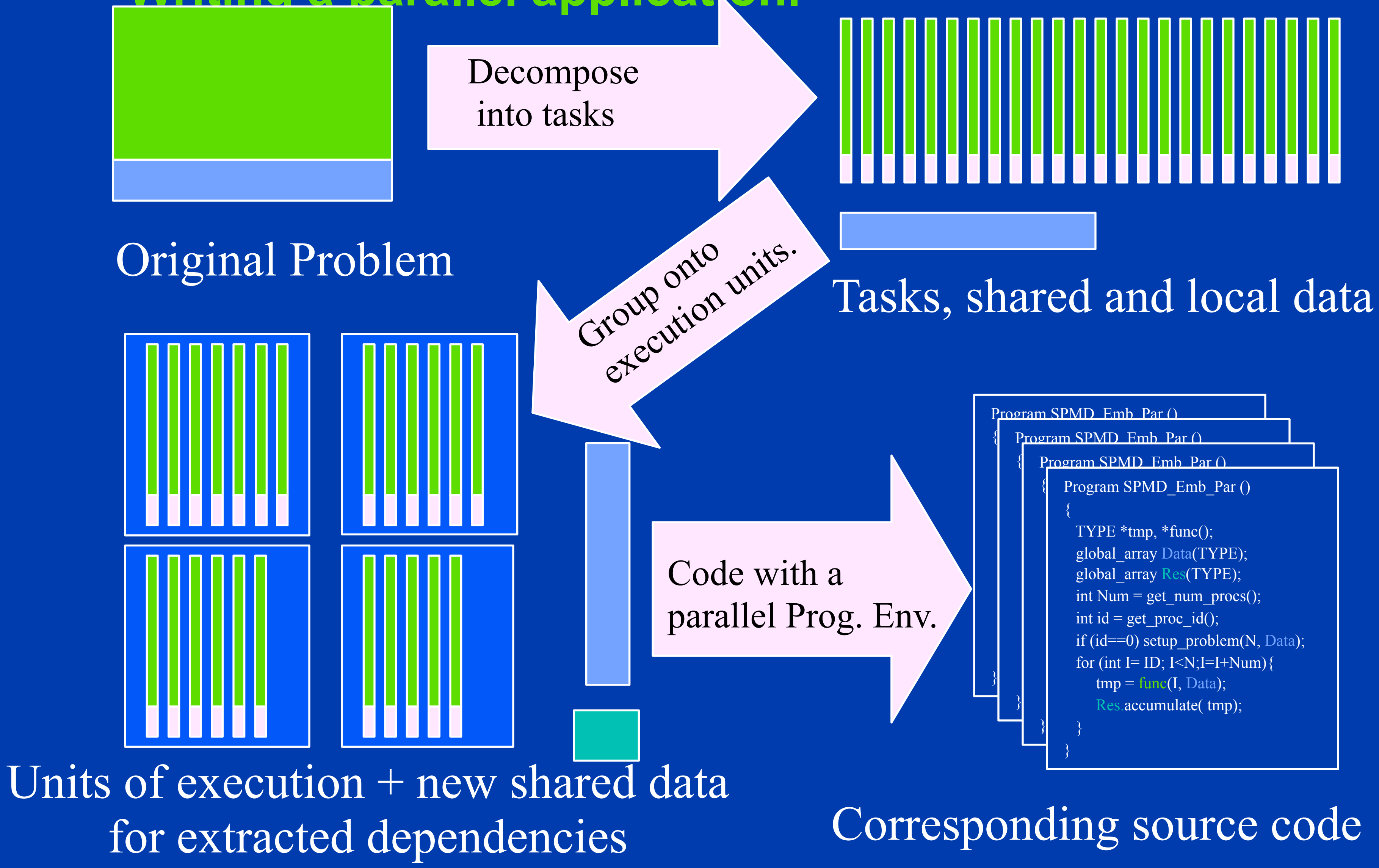
School of Electrical and Computer Engineering

Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction

Parallel Computing:

Writing a parallel application.


Decompose into tasks
Original Problem
Tasks, shared and local data
Group onto execution units.
Units of execution + new shared data for extracted dependencies
Code with a parallel Prog. Env.
Corresponding source code

```
Program SPMD_Emb_Par ()
{
  TYPE *tmp, *func();
  global_array Data(TYPE);
  global_array Res(TYPE);
  int Num = get_num_procs();
  int id = get_proc_id();
  if (id==0) setup_problem(N, Data);
  for (int I= ID; I<N;I=I+Num){
      tmp = func(I, Data);
      Res.accumulate( tmp);
  }
}
```

Parallel Computing:
Effective Standards for Portable programming

- Thread Libraries
  - Win32 API
  - POSIX threads.
- Compiler Directives
  - OpenMP - portable shared memory parallelism.
- Message Passing Libraries
  - MPI

Parallel Computing:
Effective Standards for Portable programming

- Thread Libraries
  - Win32 API
  - POSIX threads.
- Compiler Directives
  - OpenMP - portable shared memory parallelism.
- Message Passing Libraries
  - MPI

Our focus

Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- ➡ OpenMP: A quick overview
- OpenMP: A detailed introduction

OpenMP Overview:

```
C$OMP FLUSH
```

```
#pragma omp critical
```

```
C$OMP THREADPRIVATE(/ABC/)
```

```
CALL OMP_SET_NUM_THREADS(10)
```

```
C$OMP parallel do shared(a, b, c)
```

```
call omp_test_lock(jlok)
```

```
call OMP_INIT_LOCK (ilok)
```

```
C$OMP ATOMIC
```

```
C$OMP
MASTER
```

```
C$OMP  SINGLE PRIVATE(X)
```

```
setenv OMP_SCHEDULE "dynamic"
```

```
C$OMP PARALLEL DO ORDERED PRIVATE (A, B, C)
```

```
C$OMP ORDERED
```

```
C$OMP PARALLEL  REDUCTION (+: A,
B)
```

```
C$OMP
SECTIONS
```

```
#pragma omp parallel for private(A, B)
```

```
!$OMP  BARRIER
```

```
C$OMP PARALLEL COPYIN(/blk/)
```

```
C$OMP DO lastprivate(XX)
```

```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

OpenMP Overview:

```
C$OMP FLUSH
```

```
#pragma omp critical
```

```
C$OMP PARALLEL COPYIN(/blk/)
```

```
C$OMP DO lastprivate(XX)
```

```
Nthrds = OMP_GET_NUM_PROCS()
```

```
omp_set_lock(lck)
```

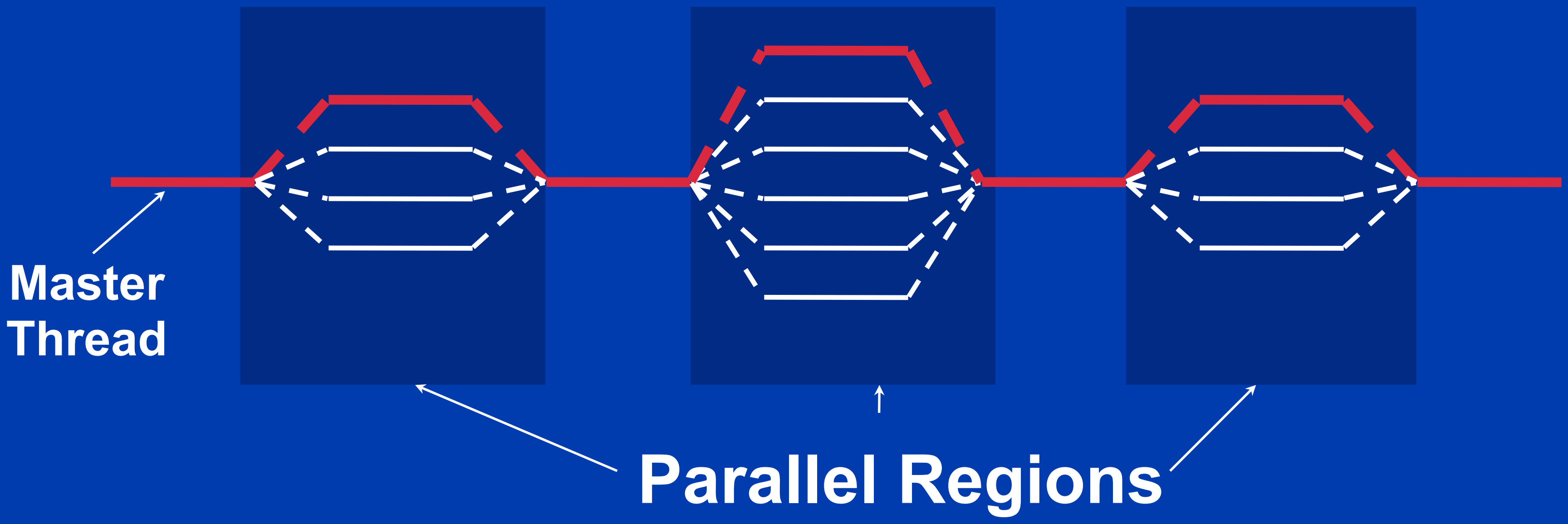
OpenMP: An API for Writing Multithreaded Applications

- A set of compiler directives and library routines for parallel application programmers
- Makes it easy to create multi-threaded (MT) programs in Fortran, C and C++
- Standardizes last 15 years of SMP practice

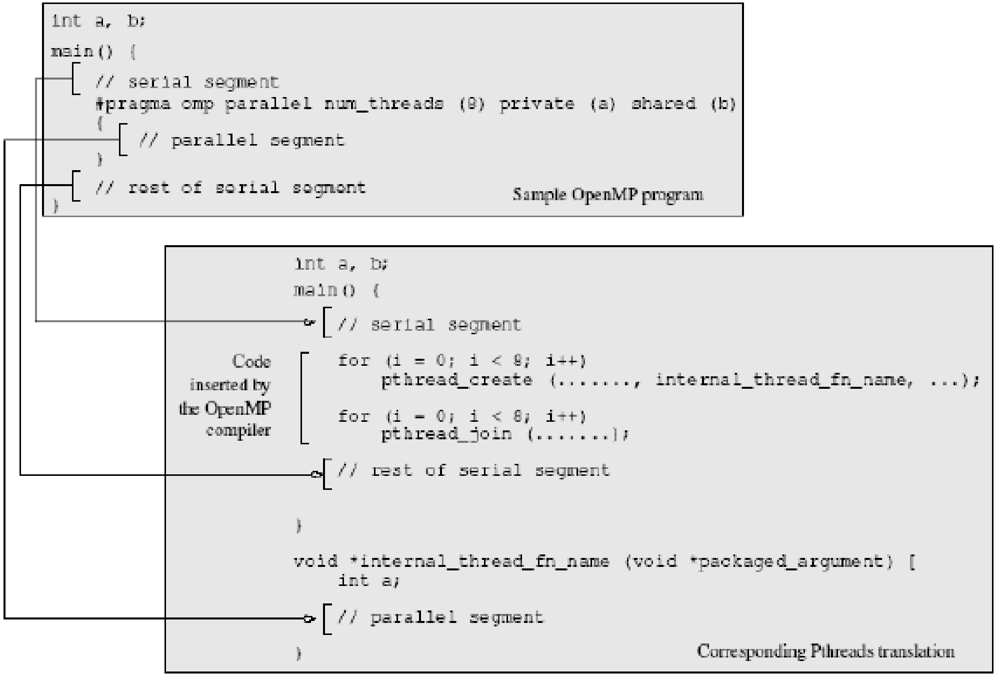
OpenMP Overview:
Programming Model

Fork-Join Parallelism:

- Master thread spawns a team of threads as needed.
- Parallelism is added incrementally: i.e. the sequential program evolves into a parallel program.



OpenMP Programming Model



```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

Sample OpenMP program

```
int a, b;
main() {
    // serial segment
    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);
    // rest of serial segment

}

void *internal_thread_fn_name (void *packaged_argument) [
    int a;
    // parallel segment
}
```

Corresponding Pthreads translation

- A sample OpenMP program along with its Pthreads translation that might be performed by an OpenMP compiler.

OpenMP Overview:
How is OpenMP typically used? (in C)

- OpenMP is usually used to parallelize loops:
  - Find your most time consuming loops.
  - Split them up between threads.

Split-up this loop between multiple threads

```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Sequential Program

```
#include “omp.h”
void main()
{
   double Res[1000];
#pragma omp parallel for
   for(int i=0;i<1000;i++) {
         do_huge_comp(Res[i]);
   }
}
```

Parallel Program

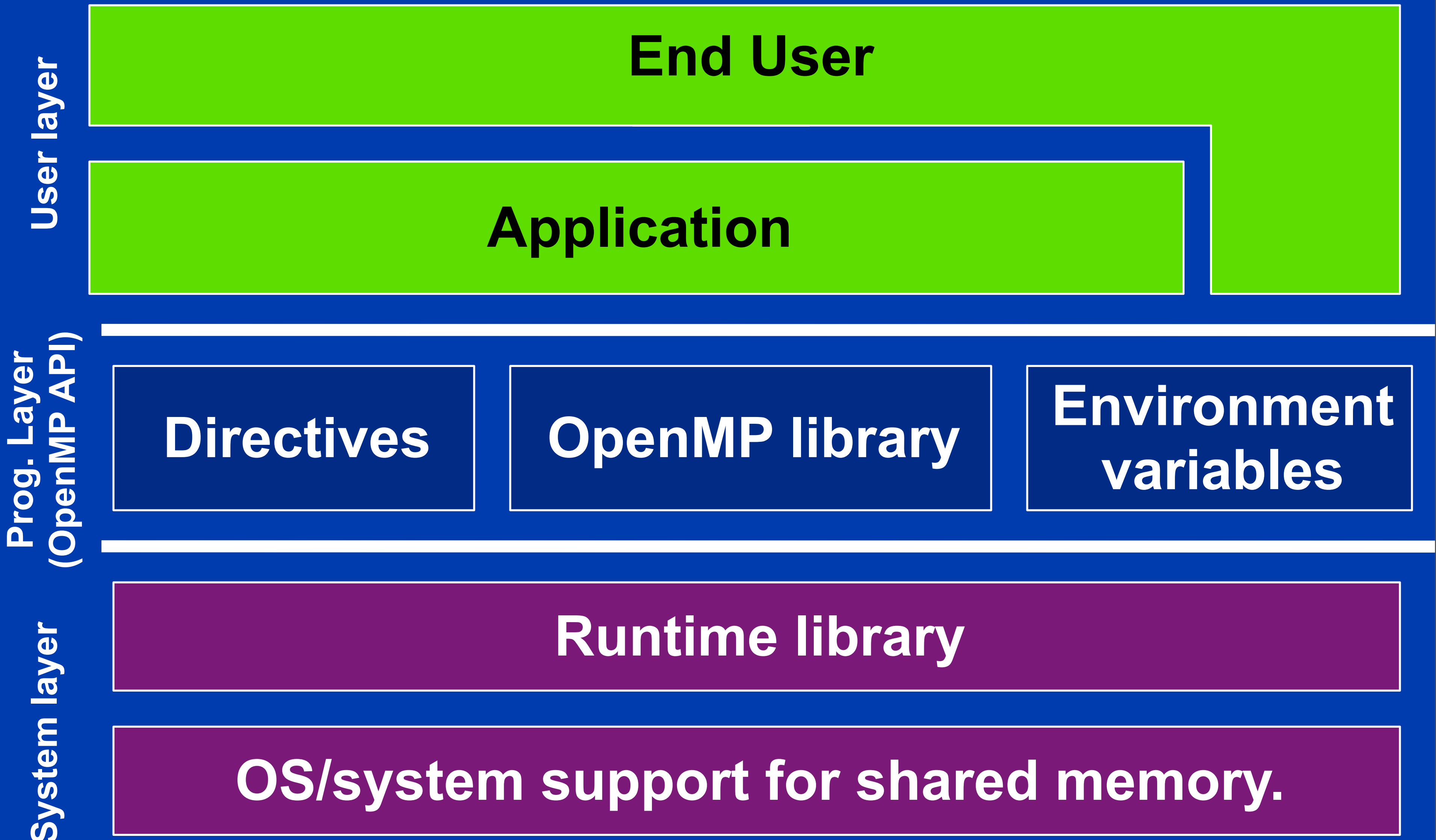
OpenMP Overview:
How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.
- Unintended sharing of data causes race conditions:
  - race condition: when the program’s outcome changes as the threads are scheduled differently.
- To control race conditions:
  - Use synchronization to protect data conflicts.
- Synchronization is expensive so:
  - Change how data is accessed to minimize the need for synchronization.

Agenda

- Setting the stage
    - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- ➡ OpenMP: A detailed introduction

OpenMP Parallel Computing Solution Stack



OpenMP:
Some syntax details to get us started

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:

    #pragma omp *construct [clause [clause]…]*
  - For Fortran, the directives take one of the forms:

    C$OMP *construct [clause [clause]…]*

    !$OMP *construct [clause [clause]…]*

    $OMP *construct [clause [clause]…]
- Include file and the OpenMP lib module

  #include “omp.h”

  use omp_lib

OpenMP:
Structured blocks (C/C++)

- Most OpenMP* constructs apply to structured blocks.
  - Structured block: a block with one point of entry at the top and one point of exit at the bottom.
  - The only "branches" allowed are STOP statements in Fortran and exit() in C/C++.

```
#pragma omp parallel
{
          int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
          if(conv(res(id)) goto more;
}
   printf(" All done \n");
```

A structured block

```
      if(go_now()) goto more;
#pragma omp parallel
{
          int id = omp_get_thread_num();
more:  res(id) = do_big_job(id);
          if(conv(res(id)) goto done;
          go to more;
}
done:     if(!really_done()) goto more;
```

<u>Not</u> A structured block

OpenMP:
Structured Block Boundaries

- In C/C++: a block is a single statement or a group of statements between brackets {}

```
#pragma omp parallel
{
    id = omp_thread_num();
    res(id) = lots_of_work(id);
}
```

```
#pragma omp for
  for(I=0;I<N;I++){
        res[I] = big_calc(I);
        A[I] = B[I] + res[I];
  }
```

- In Fortran: a block is a single statement or a group of statements between directive/end-directive pairs.

```
C$OMP PARALLEL
10    wrk(id) = garbage(id)
      res(id) = wrk(id)**2
      if(conv(res(id)) goto 10
C$OMP END PARALLEL
```

```
C$OMP PARALLEL DO
      do I=1,N
          res(I)=bigComp(I)
       end do
C$OMP END PARALLEL DO
```

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - ◆Parallel Regions
  - ◆Worksharing
  - ◆Data Environment
  - ◆Synchronization
  - ◆Runtime functions/environment variables
- OpenMP is basically the same between Fortran and C/C++

The OpenMP* API

Parallel Regions

- You create threads in OpenMP* with the "omp parallel" pragma.
- For example, To create a 4 thread Parallel region:

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID = omp_get_thread_num();
        pooh(ID,A);
}
```

Each thread executes a copy of the code within the structured block

Runtime function to request a certain number of threads

Runtime function returning a thread ID

- Each thread calls pooh(ID,A) for `ID` = 0 to 3

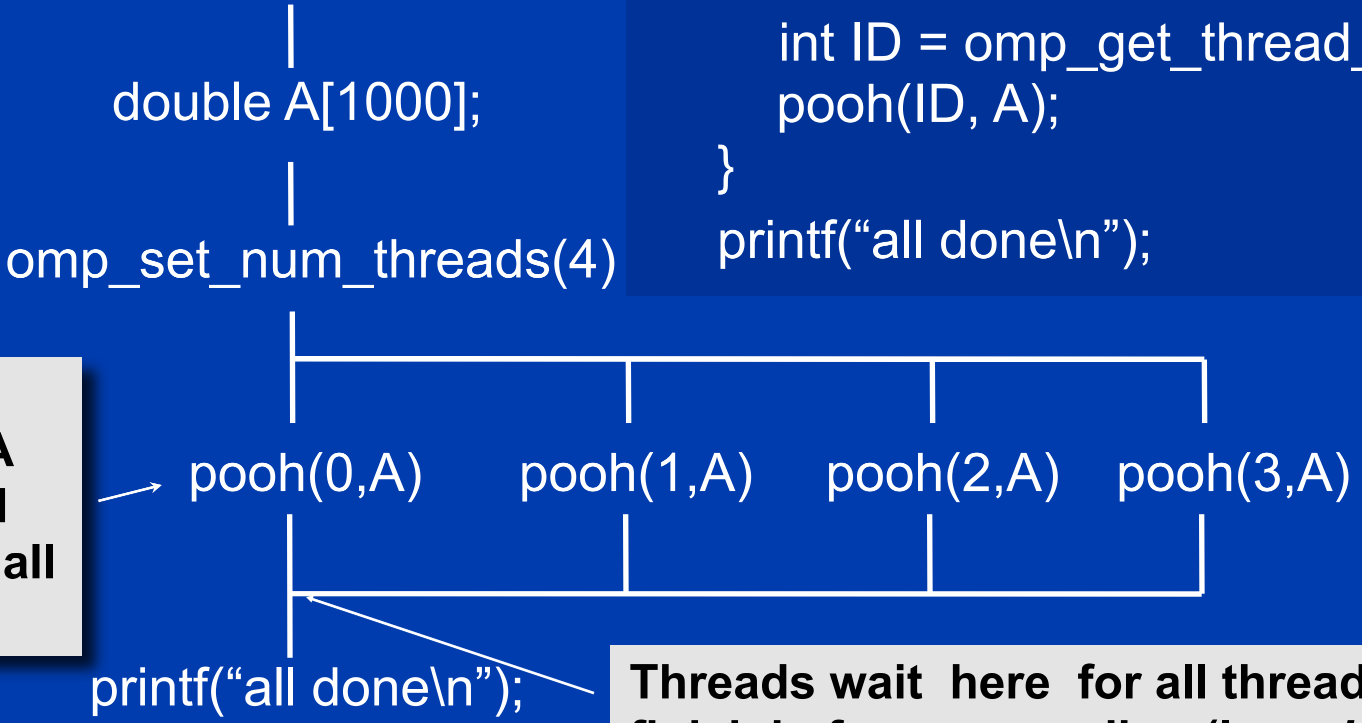
* Third party trademarks and names are the property of their respective owner.

The OpenMP* API

Parallel Regions

- **Each thread executes the same code redundantly.**

```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_get_thread_num();
    pooh(ID, A);
}
printf("all done\n");
```



A single copy of A is shared between all threads.

Threads wait here for all threads to finish before proceeding (I.e. a *barrier*)

* Third party trademarks and names are the property of their respective owner.

Exercise 1:
A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
void main()
{

    int ID = omp_get_thread_num();
    printf(“ hello(%d) ”, ID);
    printf(“ world(%d) \n”, ID);
}
```

Exercise 1:
A multi-threaded “Hello world” program

- Write a multithreaded program where each thread prints “hello world”.

```
#include “omp.h”
void main()
{
#pragma omp parallel
 {
     int ID = omp_get_thread_num();
     printf(“ hello(%d) ”, ID);
     printf(“ world(%d) \n”, ID);
   }
}
```

Sample Output:

```
hello(1) hello(0) world(1)
world(0)
hello (3) hello(2) world(3)
world(2)
```

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - ◆Parallel Regions
  - ◆Work-sharing
  - ◆Data Environment
  - ◆Synchronization
  - ◆Runtime functions/environment variables

OpenMP: Work-Sharing Constructs

- The “for” Work-Sharing construct splits up loop iterations among the threads in a team

```
#pragma omp parallel
#pragma omp for
        for (I=0;I<N;I++){
                NEAT_STUFF(I);
        }
```

By default, there is a barrier at the end of the “omp for”. Use the “nowait” clause to turn off the barrier.

Work Sharing Constructs
A motivating example

Sequential code

```
for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP parallel region

```
#pragma omp parallel
{
            int id, i, Nthrds, istart, iend;
            id = omp_get_thread_num();
            Nthrds = omp_get_num_threads();
            istart = id * N / Nthrds;
            iend = (id+1) * N / Nthrds;
            for(i=istart;I<iend;i++)   { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a work-sharing for-construct

```
#pragma omp parallel
#pragma omp for schedule(static)
            for(i=0;I<N;i++)   { a[i] = a[i] + b[i];}
```

OpenMP For/do construct:

The schedule clause

- **The schedule clause effects how loop iterations are mapped onto threads**
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size "chunk" to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size "chunk" as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.

The OpenMP* API

The schedule clause

Schedule Clause	When To Use
STATIC	Predictable and similar work per iteration
DYNAMIC	Unpredictable, highly variable work per iteration
GUIDED	Special case of dynamic to reduce scheduling overhead

* Third party trademarks and names are the property of their respective owner.

OpenMP: Work-Sharing Constructs

- The Sections work-sharing construct gives a different structured block to each thread.

```
#pragma omp parallel
#pragma omp sections
{
#pragma omp section
        X_calculation();
#pragma omp section
        y_calculation();
#pragma omp section
        z_calculation();
}
```

By default, there is a barrier at the end of the "omp sections". Use the "nowait" clause to turn off the barrier.

The OpenMP* API

Combined parallel/work-share

- OpenMP* shortcut: Put the “parallel” and the work-share on the same line

```
 double  res[MAX];  int i;
#pragma omp parallel
{
   #pragma omp for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
}
```

```
 double  res[MAX];  int i;
#pragma omp parallel for
   for (i=0;i< MAX; i++) {
       res[i] = huge();
   }
```

These are equivalent

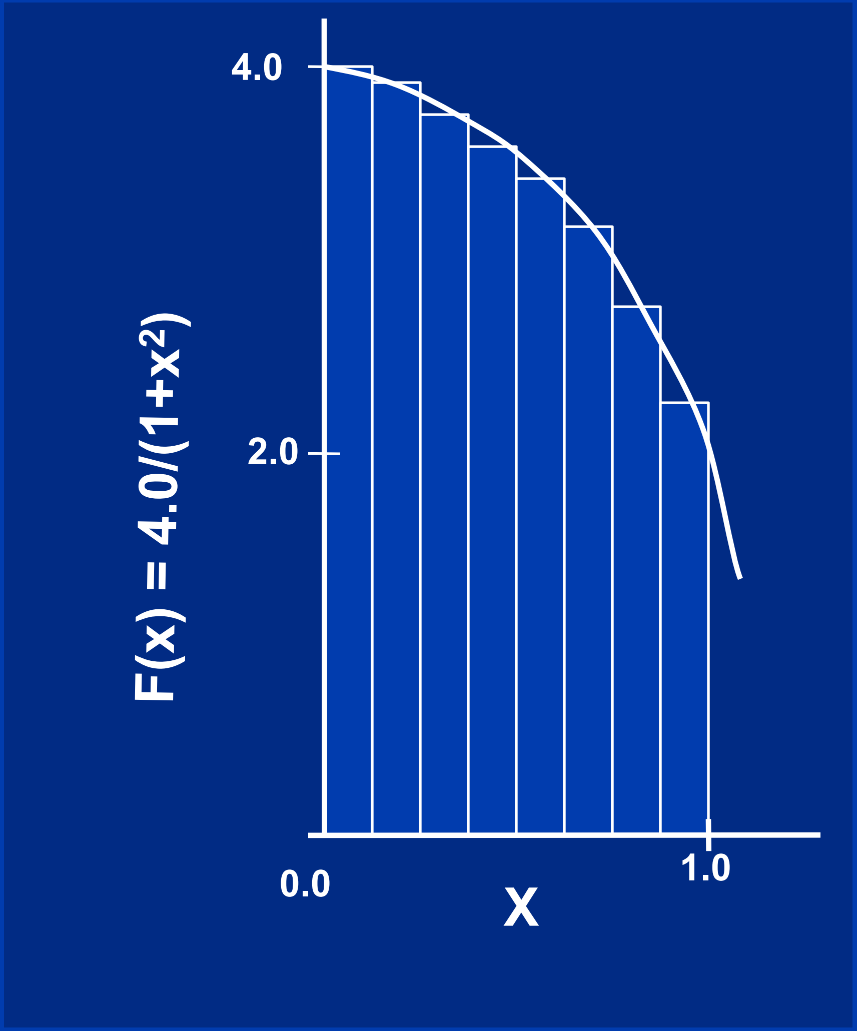
- There’s also a “parallel sections” construct.

Exercise 2:
A multi-threaded “pi” program

- On the following slide, you’ll see a sequential program that uses numerical integration to compute an estimate of PI.
- Parallelize this program using OpenMP. There are several options (do them all if you have time):
  - Do it as an SPMD program using a parallel region only.
  - Do it with a work sharing construct.
- Remember, you’ll need to make sure multiple threads don’t overwrite each other’s variables.

Our running Example: The PI program

Numerical Integration



Mathematically, we know that:

$$\int_0^1 \frac{4.0}{(1+x^2)}\, dx = \pi$$

We can approximate the integral as a sum of rectangles:

$$\sum_{i=0}^{N} F(x_i)\Delta x \approx \pi$$

Where each rectangle has width Δx and height $F(x_i)$ at the middle of interval i.

PI Program:
The sequential program

```
static long num_steps = 100000;
double step;
void main ()
{           int i;   double x, pi, sum = 0.0;

            step = 1.0/(double) num_steps;

            for (i=1;i<= num_steps; i++){
                      x = (i-0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP PI Program:

Parallel Region example (SPMD Program)

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i;      double x, pi, sum[NUM_THREADS] ={0};
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{           double x;     int id, i;
            id = omp_get_thread_num();
            int nthreads = omp_get_num_threads();
            for (i=id;i< num_steps; i=i+nthreads){
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
            }
}
            for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

SPMD Programs:

Each thread runs the same code with the thread ID selecting any thread specific behavior.

MPI: Pi program

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD) ;
}
```

OpenMP PI Program:
Work sharing construct

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{            int i;      double x, pi, sum[NUM_THREADS] ={0.0};
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{            double x;     int i, id;
             id = omp_get_thread_num();
#pragma omp for
             for (i=0;i< num_steps; i++){
                         x = (i+0.5)*step;
                         sum[id] += 4.0/(1.0+x*x);
             }
}
 for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

Solution: Win32 API, PI

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;



  start = *(int *) arg;
  step = 1.0/(double) num_steps;


  for (i=start;i<= num_steps; i=i+NUM_THREADS){
       x = (i-0.5)*step;
      sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
          thread_handles[i] = CreateThread(0, 0,
                              (LPTHREAD_START_ROUTINE) Pi,
                              &threadArg[i], 0, &threadID);
}


  WaitForMultipleObjects(NUM_THREADS,
                              thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Solution: Win32 API, PI

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;



  start = *(int *) arg;
  step = 1.0/(double) num_steps;


  for (i=start;i<= num_steps; i=i+NUM_THREADS){
      x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUp
  global_sum += sum;
  LeaveCriticalSection(&hU
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
           thread_handles[i] = CreateThread(0, 0,
                             (LPTHREAD_START_ROUTINE) Pi,
                             &threadArg[i], 0, &threadID);
}


  WaitForMultipleObjects(NUM_THREADS,
                             thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Doubles code size!

OpenMP:
Scope of OpenMP constructs

OpenMP constructs can span multiple source files.

poo.f

```
C$OMP PARALLEL
      call whoami
C$OMP END PARALLEL
```

bar.f

```
      subroutine whoami
      external omp_get_thread_num
      integer iam, omp_get_thread_num
      iam = omp_get_thread_num()
C$OMP CRITICAL
      print*,'Hello from ', iam
C$OMP END CRITICAL
      return
      end
```

+

lexical extent of parallel region

Dynamic extent of parallel region includes *lexical* extent

Orphan directives can appear outside a parallel region

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

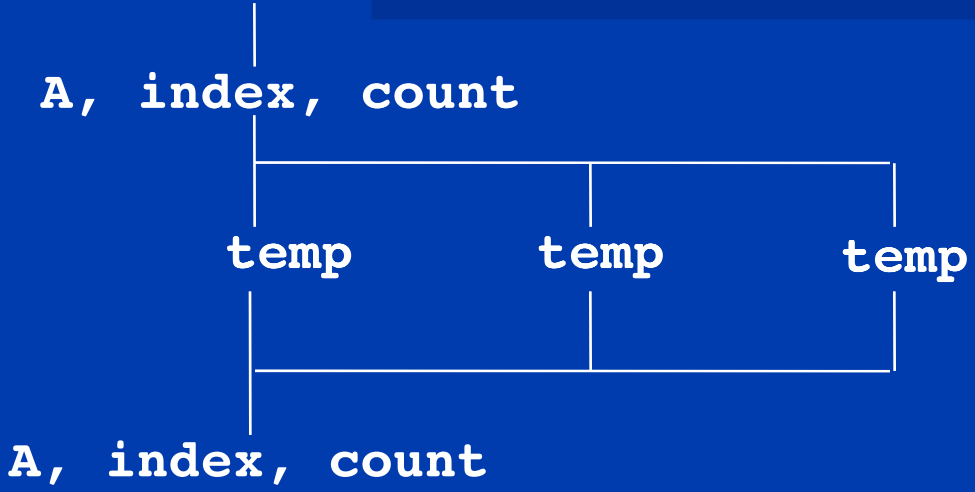
Data Environment:
Default storage attributes

- Shared Memory programming model:
  - Most variables are shared by default
- Global variables are SHARED among threads
  - Fortran: COMMON blocks, SAVE variables, MODULE variables
  - C: File scope variables, static
- But not everything is shared...
  - Stack variables in sub-programs called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE.

Data Sharing Examples

```
      program sort
      common /input/ A(10)
      integer index(10)
C$OMP PARALLEL
      call work(index)
C$OMP END PARALLEL
      print*, index(1)
```

```
subroutine work (index)
common /input/ A(10)
integer index(*)
real temp(10)
integer count
save count
      …………
```



A, index and count are shared by all threads.

temp is local to each thread

* Third party trademarks and names are the property of their respective owner.

Data Environment:
Changing storage attributes

- One can selectively change storage attributes constructs using the following clauses*
  - SHARED
  - PRIVATE
  - FIRSTPRIVATE
  - THREADPRIVATE

All the clauses on this page only apply to the *lexical extent* of the OpenMP construct.

- The value of a private inside a parallel loop can be transmitted to a global value outside the loop with:
  - LASTPRIVATE
- The default status can be modified with:
  - DEFAULT (PRIVATE | SHARED | NONE)

All data clauses apply to parallel regions and worksharing constructs except "shared" which only applies to parallel regions.

Private Clause

- private(var) creates a local copy of var for each thread.
  - The value is uninitialized
  - Private copy is *not* storage associated with the original

```
      program wrong
      IS = 0
C$OMP PARALLEL DO PRIVATE(IS)
      DO J=1,1000
          IS = IS + J
      END DO
      print *, IS
```

Regardless of initialization, IS is undefined at this point

IS was not initialized

Firstprivate Clause

- Firstprivate is a special case of private.
  - Initializes each private copy with the corresponding value from the master thread.

```
      program almost_right
      IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
      DO J=1,1000
          IS = IS + J
1000  CONTINUE
      print *, IS
```

Each thread gets its own IS with an initial value of 0

Regardless of initialization, IS is undefined at this point

Lastprivate Clause

- Lastprivate passes the value of a private from the last iteration to a global variable.

```
      program closer
      IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP+ LASTPRIVATE(IS)
      DO J=1,1000
          IS = IS + J
1000  CONTINUE
      print *, IS
```

Each thread gets its own IS with an initial value of 0

IS is defined as its value at the last iteration (I.e. for J=1000)

OpenMP:
A data environment test

- Here’s an example of PRIVATE and FIRSTPRIVATE

```
          variables A,B, and C = 1
C$OMP PARALLEL PRIVATE(B)
C$OMP& FIRSTPRIVATE(C)
```

OpenMP:
A data environment test

- Here's an example of PRIVATE and FIRSTPRIVATE

```
        variables A,B, and C = 1
C$OMP PARALLEL PRIVATE(B)
C$OMP& FIRSTPRIVATE(C)
```

- Inside this parallel region ...
  - "A" is shared by all threads; equals 1
  - "B" and "C" are local to each thread.
    - B's initial value is undefined
    - C's initial value equals 1
- Outside this parallel region ...
  - The values of "B" and "C" are undefined.

Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to specify)
- To change default: DEFAULT(PRIVATE)
  - *each* variable in *static* extent of the parallel region is made private as if specified in a private clause
  - mostly saves typing
- DEFAULT(NONE)*:* *no* default for variables in static extent. Must list storage attribute for each variable in static extent

Only the Fortran API supports default(private).

C/C++ only has default(shared) or default(none).

Default Clause Example

```
            itotal = 1000
C$OMP PARALLEL PRIVATE(np, each)
      np = omp_get_num_threads()
      each = itotal/np
      .........
C$OMP END PARALLEL
```

```
      itotal = 1000
C$OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)
      np = omp_get_num_threads()
      each = itotal/np
      .........
C$OMP END PARALLEL
```

These two codes are equivalent

Threadprivate

- Makes global data private to a thread
  - Fortran: COMMON blocks
  - C: File scope and static variables
- Different from making them PRIVATE
  - with PRIVATE global variables are masked.
  - THREADPRIVATE preserves global scope within each thread
- Threadprivate variables can be initialized using COPYIN or by using DATA statements.

A threadprivate example

Consider two different routines called within a parallel region.

```
      subroutine poo
      parameter (N=1000)
      common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
      do i=1, N
        B(i)= const* A(i)
      end do
      return
      end
```

```
      subroutine bar
      parameter (N=1000)
      common/buf/A(N),B(N)
C$OMP THREADPRIVATE(/buf/)
      do i=1, N
        A(i) = sqrt(B(i))
      end do
      return
      end
```

Because of the threadprivate construct, each thread executing these routines has its own copy of the common block /buf/.

Copyprivate

You initialize threadprivate data using a copyprivate clause.

```
      parameter (N=1000)
      common/buf/A(N)
C$OMP THREADPRIVATE(/buf/)

C Initialize the A array
      call init_data(N,A)

C$OMP PARALLEL COPYPRIVATE(A)

 … Now each thread sees threadprivate array A initialied
 … to the global value set in the subroutine init_data()

C$OMP END PARALLEL

end
```

OpenMP: Reduction

- Another clause that effects the way variables are shared:

  reduction (op : list)

- The variables in “list” must be shared in the enclosing parallel region.
- Inside a parallel or a work-sharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”).
  - Compiler finds standard reduction expressions containing “op” and uses them to update the local copy.
  - Local copies are reduced into a single value and combined with the original global value.

OpenMP:
Reduction example

```
#include <omp.h>
#define NUM_THREADS 2
void main ()
{
        int i;
        double ZZ, func(), res=0.0;
        omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:res) private(ZZ)
        for (i=0; i< 1000; i++){
              ZZ = func(I);
              res = res + ZZ;
        }
}
```

OpenMP: Reduction example

- Remember the code we used to demo private, firstprivate and lastprivate.

```
      program closer
      IS = 0
      DO J=1,1000
          IS = IS + J
1000  CONTINUE
      print *, IS
```

OpenMP: Reduction example

- Remember the code we used to demo private, firstprivate and lastprivate.

```
      program closer
      IS = 0
      DO J=1,1000
          IS = IS + J
1000  CONTINUE
      print *, IS
```

- Here is the correct way to parallelize this code.

```
      program closer
      IS = 0
#pragma omp parallel for reduction(+:IS)
      DO J=1,1000
          IS = IS + J
1000  CONTINUE
      print *, IS
```

OpenMP:
Reduction operands/initial-values

- A range of associative operands can be used with reduction:
- Initial values are the ones that make sense mathematically.

Operand	Initial value
+	0
*	1
-	0
.AND.	All 1's

Operand	Initial value
.OR.	0
MAX	1
MIN	0
//	All 1's

Exercise 3:
A multi-threaded "pi" program

- Return to your "pi" program and this time, use private, reduction and a work-sharing construct to parallelize it.
- See how similar you can make it to the original sequential program.

OpenMP PI Program :
Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{            int i;     double x, pi, sum = 0.0;
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
             for (i=1;i<= num_steps; i++){
                        x = (i-0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
             }
             pi = step * sum;
}
```

OpenMP PI Program : Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{           int i;     double x, pi, sum = 0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
            for (i=1;i<= num_steps; i++){
                      x = (i-0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP adds 2 to 4 lines of code

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - ◆Parallel Regions
  - ◆Worksharing
  - ◆Data Environment
  - ◆Synchronization
  - ◆Runtime functions/environment variables

OpenMP: Synchronization

- OpenMP has the following constructs to support synchronization:
  - critical section
  - atomic
  - barrier
  - flush
  - ordered
  - single
  - master

We will save flush for the advanced OpenMP tutorial.

We discuss this here, but it really isn't a synchronization construct. It's a work-sharing construct that may include synchronization.

We discus this here, but it really isn't a synchronization construct.

The OpenMP* API

Synchronization – critical section (in C/C++)

- Only one thread at a time can enter a **critical** section.

```
float res;
#pragma omp parallel
{    float B;   int i;
     #pragma omp for
     for(i=0;i<niters;i++){
          B =  big_job(i);
#pragma omp critical
               consum (B, RES);
     }
}
```

Threads wait their turn – only one at a time calls consum()

* Third party trademarks and names are the property of their respective owner.

OpenMP: Synchronization

- **Atomic** is a special case of a critical section that can be used for certain simple statements.
- It applies only to the update of a memory location (the update of X in the following example)

```
C$OMP PARALLEL PRIVATE(B)
        B =  DOIT(I)
tmp = big_ugly();

 C$OMP ATOMIC
        X = X + temp

C$OMP END PARALLEL
```

OpenMP: Synchronization

- **Barrier**: Each thread waits until all threads arrive.

```
#pragma omp parallel shared (A, B, C) private(id)
{
        id=omp_get_thread_num();
        A[id] = big_calc1(id);
#pragma omp barrier
#pragma omp for
        for(i=0;i<N;i++){C[i]=big_calc3(I,A);}
#pragma omp for nowait
        for(i=0;i<N;i++){ B[i]=big_calc2(C,  i); }
        A[id] = big_calc3(id);
}
```

implicit barrier at the end of a for work-sharing construct

no implicit barrier due to nowait

implicit barrier at the end of a parallel region

OpenMP: Synchronization

- The ordered construct enforces the sequential order for a block.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res += consum(tmp);
        }
```

OpenMP: ~~Synchronization~~

- The **master** construct denotes a structured block that is only executed by the master thread. The other threads just skip it (no synchronization is implied).

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp master
        {    exchange_boundaries();   }
#pragma barrier
        do_many_other_things();
}
```

OpenMP: ~~Synchronization~~ work-share

- The single construct denotes a block of code that is executed by only one thread.
- A barrier is implied at the end of the single block.

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp single
        {    exchange_boundaries();   }
        do_many_other_things();
}
```

OpenMP:
Implicit synchronization

- Barriers are implied on the following OpenMP constructs:

```
end parallel
end do  (except when nowait is used)
end sections (except when nowait is used)
end single (except when nowait is used)
```

OpenMP PI Program:
Parallel Region example (SPMD Program)

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i;      double x, pi, sum[NUM_THREADS];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS)
#pragma omp parallel
{           double x;     int id;
            id = omp_get_thread_num();
            for (i=id, sum[id]=0.0;i< num_steps; i=i+NUM_THREADS){
                        x = (i+0.5)*step;
                        sum[id] += 4.0/(1.0+x*x);
            }
}
            for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

Performance would be awful due to false sharing of the sum array.

OpenMP PI Program:

use a critical section to avoid the array

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i;      double  x, sum, pi=0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS)
#pragma omp parallel private (x, sum)
{
            id = omp_get_thread_num();
            for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
                  x = (i+0.5)*step;
                  sum += 4.0/(1.0+x*x);
            }
#pragma omp critical
            pi += sum
}
}
```

No array, so no false sharing. However, poor scaling with the number of threads

OpenMP: Contents

- OpenMP's constructs fall into 5 categories:
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables

OpenMP: Library routines: Part 1

- Runtime environment routines:
  - Modify/Check the number of threads
    - omp_set_num_threads()
    - omp_get_num_threads()
    - omp_get_thread_num()
    - omp_get_max_threads()
  - Are we in a parallel region?
    - omp_in_parallel()
  - How many processors in the system?
    - omp_num_procs()

OpenMP: Library Routines

- To fix the number of threads used in a program, (1) set the number threads, then (4) save the number you got.

```
#include <omp.h>
void main()
{   int num_threads;
    omp_set_num_threads( omp_num_procs() );
#pragma omp parallel
    {     int id=omp_get_thread_num();
#pragma omp single
              num_threads = omp_get_num_threads();
           do_lots_of_stuff(id);
     }
}
```

Request as many threads as you have processors.

Protect this op since Memory stores are not atomic

OpenMP: Environment Variables: Part 1

- Control how "omp for schedule(RUNTIME)" loop iterations are scheduled.
  - OMP_SCHEDULE "schedule[, chunk_size]"
- Set the default number of threads to use.
  - OMP_NUM_THREADS *int_literal*

Summary

- OpenMP is:
  - A great way to write parallel code for shared memory machines.
  - A very simple approach to parallel programming.
  - Your gateway to special, painful errors (race conditions).

Reference Material on OpenMP*

OpenMP Homepage www.openmp.org:
The primary source of information about OpenMP and its development.

Books:
Parallel programming in OpenMP, Chandra, Rohit, San Francisco, Calif. : Morgan Kaufmann ; London : Harcourt, 2000, ISBN: 1558606718

Research papers:
Sosa CP, Scalmani C, Gomperts R, Frisch MJ. Ab initio quantum chemistry on a ccNUMA architecture using OpenMP. III. Parallel Computing, vol.26, no.7-8, July 2000, pp.843-56. Publisher: Elsevier, Netherlands.

Bova SW, Breshears CP, Cuicchi C, Demirbilek Z, Gabb H. Nesting OpenMP in an MPI application. Proceedings of the ISCA 12th International Conference. Parallel and Distributed Systems. ISCA. 1999, pp. 566-71. Cary, NC, USA.

Gonzalez M, Serra A, Martorell X, Oliver J, Ayguade E, Labarta J, Navarro N. Applying interposition techniques for performance analysis of OPENMP parallel applications. Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000. IEEE Comput. Soc. 2000, pp.235-40. Los Alamitos, CA, USA.

J. M. Bull and M. E. Kambites. JOMPan OpenMP-like interface for Java. Proceedings of the ACM 2000 conference on Java Grande, 2000, Pages 44 - 53.

Chapman B, Mehrotra P, Zima H. Enhancing OpenMP with features for locality control. Proceedings of Eighth ECMWF Workshop on the Use of Parallel Processors in Meteorology. Towards Teracomputing. World Scientific Publishing. 1999, pp.301-13. Singapore.

Cappello F, Richard O, Etiemble D. Performance of the NAS benchmarks on a cluster of SMP PCs using a parallelization of the MPI programs with OpenMP. Parallel Computing Technologies. 5th International Conference, PaCT-99. Proceedings (Lecture Notes in Computer Science Vol.1662). Springer-Verlag. 1999, pp.339-50. Berlin, Germany.

Couturier R, Chipot C. Parallel molecular dynamics using OPENMP on a shared memory machine. Computer Physics Communications, vol.124, no.1, Jan. 2000, pp.49-59. Publisher: Elsevier, Netherlands.

Bova SW, Breshearsz CP, Cuicchi CE, Demirbilek Z, Gabb HA. Dual-level parallel analysis of harbor wave response using MPI and OpenMP. International Journal of High Performance Computing Applications, vol.14, no.1, Spring 2000, pp.49-64. Publisher: Sage Science Press, USA.

Scherer A, Honghui Lu, Gross T, Zwaenepoel W. Transparent adaptive parallelism on NOWS using OpenMP. ACM. Sigplan Notices (Acm Special Interest Group on Programming Languages), vol.34, no.8, Aug. 1999, pp.96-106. USA.

Ayguade E, Martorell X, Labarta J, Gonzalez M, Navarro N. Exploiting multiple levels of parallelism in OpenMP: a case study. Proceedings of the 1999 International Conference on Parallel Processing. IEEE Comput. Soc. 1999, pp.172-80. Los Alamitos, CA, USA.

Mattson, T.G. An Introduction to OpenMP 2.0, Proceedings 3rd International Symposium on High Performance Computing, Lecture Notes in Computer Science, Number 1940, Springer, 2000 pp. 384-390, Tokyo Japan.

Honghui Lu, Hu YC, Zwaenepoel W. OpenMP on networks of workstations. Proceedings of ACM/IEEE SC98: 10th Anniversary. High Performance Networking and Computing Conference (Cat. No. RS00192). IEEE Comput. Soc. 1998, pp.13 pp.. Los Alamitos, CA, USA.

Throop J. OpenMP: shared-memory parallelism from the ashes. Computer, vol.32, no.5, May 1999, pp. 108-9. Publisher: IEEE Comput. Soc, USA.

Hu YC, Honghui Lu, Cox AL, Zwaenepoel W. OpenMP for networks of SMPs. Proceedings 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing. IPPS/SPDP 1999. IEEE Comput. Soc. 1999, pp.302-10. Los Alamitos, CA, USA.

Parallel Programming with Message Passing and Directives; Steve W. Bova, Clay P. Breshears, Henry Gabb, Rudolf Eigenmann, Greg Gaertner, Bob Kuhn, Bill Magro, Stefano Salvini; SIAM News, Volume 32, No 9, Nov. 1999.

Still CH, Langer SH, Alley WE, Zimmerman GB. Shared memory programming with OpenMP. Computers in Physics, vol.12, no.6, Nov.-Dec. 1998, pp.577-84. Publisher: AIP, USA.

Chapman B, Mehrotra P. OpenMP and HPF: integrating two paradigms. [Conference Paper] Euro-Par'98 Parallel Processing. 4th International Euro-Par Conference. Proceedings. Springer-Verlag. 1998, pp.650-8. Berlin, Germany.

Dagum L, Menon R. OpenMP: an industry standard API for shared-memory programming. IEEE Computational Science & Engineering, vol.5, no.1, Jan.-March 1998, pp.46-55. Publisher: IEEE, USA.

Clark D. OpenMP: a parallel standard for the masses. IEEE Concurrency, vol.6, no.1, Jan.-March 1998, pp. 10-12. Publisher: IEEE, USA.

Extra Slides
A series of parallel pi programs

Some OpenMP Commands to support Exercises

PI Program: an example

```
static long num_steps = 100000;
double step;
void main ()
{         int i;   double x, pi, sum = 0.0;

          step = 1.0/(double) num_steps;

          for (i=1;i<= num_steps; i++){
                   x = (i-0.5)*step;
                   sum = sum + 4.0/(1.0+x*x);
          }
          pi = step * sum;
}
```

Parallel Pi Program

- Let’s speed up the program with multiple threads.
- Consider the Win32 threads library:
  - Thread management and interaction is explicit.
  - Programmer has full control over the threads

Solution: Win32 API, PI

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;



  start = *(int *) arg;
  step = 1.0/(double) num_steps;


  for (i=start;i<= num_steps; i=i+NUM_THREADS){
      x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUpdateMutex);
  global_sum += sum;
  LeaveCriticalSection(&hUpdateMutex);
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
            thread_handles[i] = CreateThread(0, 0,
                                   (LPTHREAD_START_ROUTINE) Pi,
                                   &threadArg[i], 0, &threadID);
}


  WaitForMultipleObjects(NUM_THREADS,
                                   thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Solution: Win32 API, PI

```
#include <windows.h>
#define NUM_THREADS 2
HANDLE thread_handles[NUM_THREADS];
CRITICAL_SECTION hUpdateMutex;
static long num_steps = 100000;
double step;
double global_sum = 0.0;

void Pi (void *arg)
{
   int i, start;
  double x, sum = 0.0;



  start = *(int *) arg;
  step = 1.0/(double) num_steps;


  for (i=start;i<= num_steps; i=i+NUM_THREADS){
      x = (i-0.5)*step;
     sum = sum + 4.0/(1.0+x*x);
  }
  EnterCriticalSection(&hUp
  global_sum += sum;
  LeaveCriticalSection(&hU
}
```

```
void main ()
{
  double pi; int i;
  DWORD threadID;
  int threadArg[NUM_THREADS];

  for(i=0; i<NUM_THREADS; i++)   threadArg[i] = i+1;

  InitializeCriticalSection(&hUpdateMutex);

  for (i=0; i<NUM_THREADS; i++){
            thread_handles[i] = CreateThread(0, 0,
                              (LPTHREAD_START_ROUTINE) Pi,
                              &threadArg[i], 0, &threadID);
}


  WaitForMultipleObjects(NUM_THREADS,
                              thread_handles, TRUE,INFINITE);

  pi = global_sum * step;

  printf(" pi is %f \n",pi);
}
```

Doubles code size!

Solution: Keep it simple

Threads libraries:

- Pro: Programmer <u>has</u> control over everything
- Con: Programmer <u>must</u> control everything

$$\text{Full control} \Longrightarrow \text{Increased complexity} \longrightarrow \text{Programmers scared away}$$

Solution: Keep it simple

Threads libraries:

- Pro: Programmer has control over everything
- Con: Programmer must control everything

Full control → Increased complexity → Programmers scared away

Sometimes a simple evolutionary approach is better

OpenMP PI Program:
Parallel Region example (SPMD Program)

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i;      double x, pi, sum[NUM_THREADS] = {0.0};
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{           double x;    int i, id;
            id = omp_get_thraead_num();
            for (i=id;i< num_steps; i=i+NUM_THREADS){
                      x = (i+0.5)*step;
                      sum[id] += 4.0/(1.0+x*x);
            }
}
            for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

SPMD Programs:

Each thread runs the same code with the thread ID selecting any thread specific behavior.

OpenMP PI Program:
Work sharing construct

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{            int i;      double x, pi, sum[NUM_THREADS] = {0.0};
             step = 1.0/(double) num_steps;
             omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{            double x;     int i, id;
             id = omp_get_thraead_num();
#pragma omp for
             for (i=id;i< num_steps; i++){
                         x = (i+0.5)*step;
                         sum[id] += 4.0/(1.0+x*x);
             }
}            for(i=0, pi=0.0;i<NUM_THREADS;i++)pi += sum[i] * step;
}
```

OpenMP PI Program:
private clause and a critical section

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i;      double  x, sum, pi=0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (x, sum,i)
{
            id = omp_get_thread_num();
            for (i=id,sum=0.0;i< num_steps;i=i+NUM_THREADS){
                        x = (i+0.5)*step;
                        sum += 4.0/(1.0+x*x);
            }
#pragma omp critical
            pi += sum * step;
}
}
```

Note: We didn't need to create an array to hold local sums or clutter the code with explicit declarations of "x" and "sum".

OpenMP PI Program :
Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;         double step;
#define NUM_THREADS 2
void main ()
{           int i;     double x, pi, sum = 0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
            for (i=1;i<= num_steps; i++){
                        x = (i-0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP PI Program :
Parallel for with a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
#define NUM_THREADS 2
void main ()
{           int i;    double x, pi, sum = 0.0;
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel for reduction(+:sum) private(x)
            for (i=1;i<= num_steps; i++){
                      x = (i-0.5)*step;
                      sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

OpenMP adds 2 to 4 lines of code

MPI: Pi program

```
#include <mpi.h>
void main (int argc, char *argv[])
{
        int i, my_id, numprocs;  double x, pi, step, sum = 0.0 ;
        step = 1.0/(double) num_steps ;
        MPI_Init(&argc, &argv) ;
        MPI_Comm_Rank(MPI_COMM_WORLD, &my_id) ;
        MPI_Comm_Size(MPI_COMM_WORLD, &numprocs) ;
        my_steps = num_steps/numprocs ;
        for (i=my_id*my_steps; i<(my_id+1)*my_steps ; i++)
        {
                x = (i+0.5)*step;
                sum += 4.0/(1.0+x*x);
        }
        sum *= step ;
        MPI_Reduce(&sum, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                        MPI_COMM_WORLD) ;
}
```