Parallel Programming Principle and Practice

Lecture 12-1 —parallel programming：

Shared Memory Programming OpenMP

Outline

- A Brief History of Parallel Languages
- OpenMP
  - Shared Memory Programming Model
  - Parallel Programming with Threads
  - OpenMP Overview
  - Parallel programming with OpenMP
    - ✓ Creating Threads
    - ✓ Parallel Loops
    - ✓ Synchronization
    - ✓ Data Environment
    - ✓ Tasks
    - ✓ Example: Recursive Matrix Multiplication

parallel programming

A Brief History of Parallel Languages

A Brief History of Parallel Languages

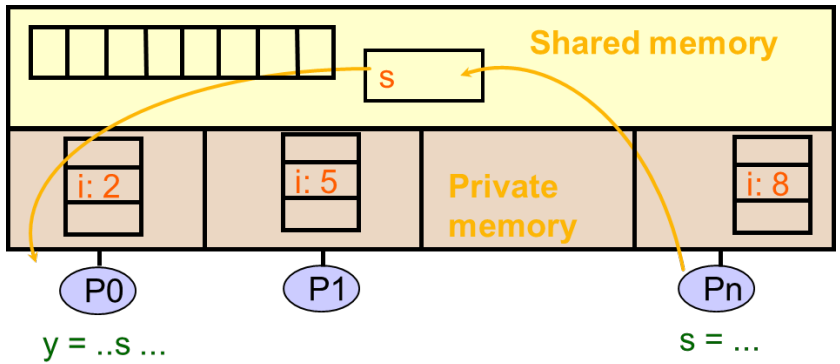
- When SIMD machines were king
  - Data parallel languages popular and successful (**CMF**, ***Lisp**, **C***, …)
  - Irregular data (sparse mat-vec multiply OK), but irregular computation (divide and conquer, adaptive meshes, etc.) less clear
- When shared memory multiprocessors (SMPs) were king
  - Shared memory models, e.g., **Posix Threads**, **OpenMP**, are popular
- When clusters took over
  - Message Passing (**MPI**) became dominant
- With the addition of accelerators
  - **OpenACC**, **CUDA** were added
- In Cloud Computing
  - **Hadoop**, **SPARK**, …

OpenMP

Shared Memory Programming Model

Recall Programming Model : Shared Memory

- Program is a collection of threads of control
  - Can be created dynamically, mid-execution, in some languages
- Each thread has a set of private variables, e.g., local stack variables
- Also a set of shared variables, e.g., static variables, shared common blocks, or global heap
  - Threads communicate implicitly by writing and reading shared variables
  - Threads coordinate by synchronizing on shared variables



OpenMP

Parallel Programming with Threads

Forking Posix Threads

Signature:

```
int pthread_create(pthread_t *,
                   const pthread_attr_t *,
                   void * (*)(void *),
                   void *);
```

Example call:

```
errcode = pthread_create(&thread_id; &thread_attribute
                         &thread_fun; &fun_arg);
```

- thread_id is the thread id or handle (used to halt, etc.)
- thread_attribute various attributes
  - Standard default values obtained by passing a NULL pointer
  - Sample attributes: minimum stack size, priority
- thread_fun the function to be run (takes and returns void*)
- fun_arg an argument can be passed to thread_fun when it starts
- errorcode will be set nonzero if the create operation fails

Example: “Simple” Threading Example

```
void* SayHello(void *foo) {
  printf( "Hello, world!\n" );
  return NULL;
}

int main() {
  pthread_t threads[16];
  int tn;
  for(tn=0; tn<16; tn++) {
    pthread_create(&threads[tn], NULL, SayHello, NULL);
  }
  for(tn=0; tn<16 ; tn++) {
    pthread_join(threads[tn], NULL);
  }
  return 0;
}
```

Compile using gcc –lpthread

Basic Types of Synchronization: Mutexes

Mutexes -- mutual exclusion aka locks

- threads are working mostly independently
- need to access common data structure

```
lock *l = alloc_and_init();    /* shared */
acquire(l);
 access data
release(l);
```

- Locks only affect processors using them
  - ✓ If a thread accesses the data without doing the acquire/release, locks by others will not help
- Java and other languages have lexically scoped synchronization, i.e., synchronized methods/blocks
- Semaphores generalize locks to allow k threads simultaneous access; good for limited resources

Example: Mutexes in POSIX Threads

- To create a mutex:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
// or pthread_mutex_init(&amutex, NULL);
```

- To use it:

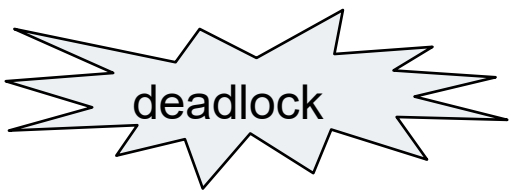
```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to problems:

thread1	thread2
lock(a)	lock(b)
lock(b)	lock(a)



- Deadlock results if both threads acquire one of their locks, so that neither can acquire the second

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages
  - Familiar language for most of program
  - Ability to shared data is convenient
- Pitfalls
  - Overhead of thread creation is high
  - Data race bugs are very nasty to find
  - Deadlocks are usually easier
- ***OpenMP*** is commonly used today as an alternative
  - ***Helps with some of these pitfalls***, but doesn't make them disappear

OpenMP

OpenMP Overview

What is OpenMP?

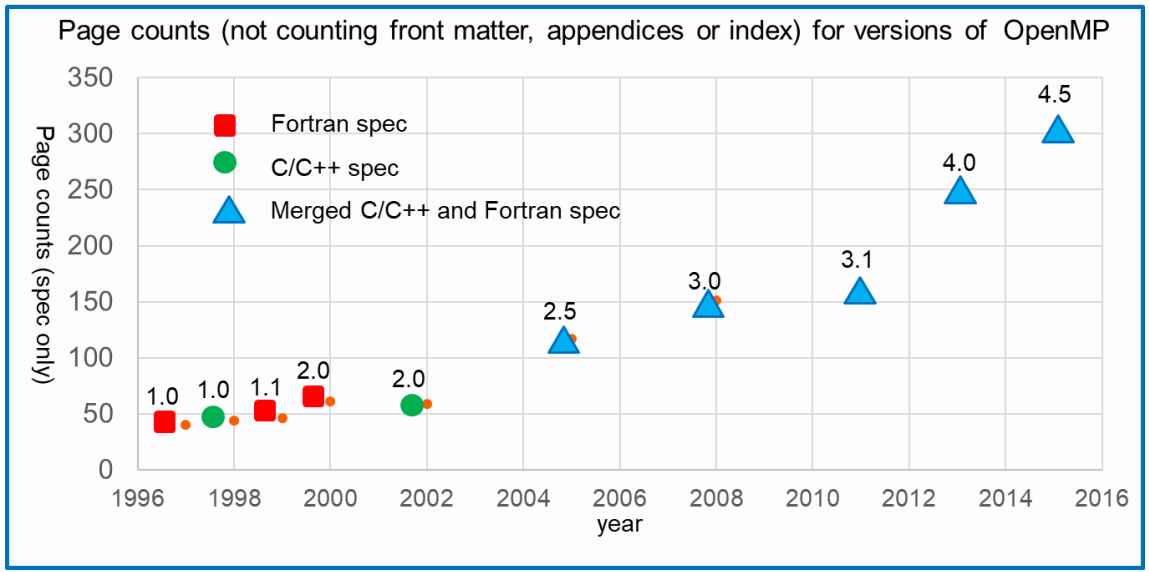
- OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared-memory multiprocessing programming in C, C++, and Fortran, on many platforms, instruction-set architectures and operating systems, including Solaris, AIX, FreeBSD, HP-UX, Linux, macOS, and Windows
- It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior
- Motivation: capture common usage and simplify programming
- OpenMP Architecture Review Board (ARB)
  - A nonprofit organization that controls the OpenMP Spec
  - Latest spec: OpenMP 5.2 (Nov. 2021)

A Programmer's View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Requires compiler support (C, C++ or Fortran)
- **OpenMP will**
  - Allow a programmer to separate a program into *serial regions* and *parallel regions*, rather than P concurrently-executing threads
  - Hide stack management
  - Provide synchronization constructs
- **OpenMP will not**
  - Parallelize automatically
  - Guarantee speedup

The growth of complexity in OpenMP

- OpenMP started out in 1997 as a simple interface for the application programmers more versed in their area of science than computer science
- The complexity has grown considerably over the years!



OpenMP 5.2 (Nov 2021) is actually **669 pages**!

✓ The complexity of the full spec is overwhelming, so we focus on the 16 constructs most OpenMP programmers restrict themselves to … the so called "**OpenMP Common Core**"

OpenMP basic syntax

- Most of the constructs in OpenMP are compiler directives

C and C++	Fortran
Compiler directives	
#pragma omp construct [clause [clause]…]	*!$OMP construct [clause [clause] …]*
Example	
#pragma omp parallel private(x) *{* *}*	*!$OMP PARALLEL* *!$OMP END PARALLEL*
Function prototypes and types:	
#include <omp.h>	*use OMP_LIB*

- Most OpenMP* constructs apply to a "structured block"
  - Structured block: a block of one or more statements with one point of entry at the top and one point of exit at the bottom
  - It's OK to have an exit() within the structured block

Example: Hello world in OpenMP

- Write a program that prints “hello world”

```
#include<stdio.h>
int main()
{




    printf(“ hello ”);
    printf(“ world \n”);

}
```

Example: Hello world in OpenMP

- Write a multithreaded program that prints “hello world”

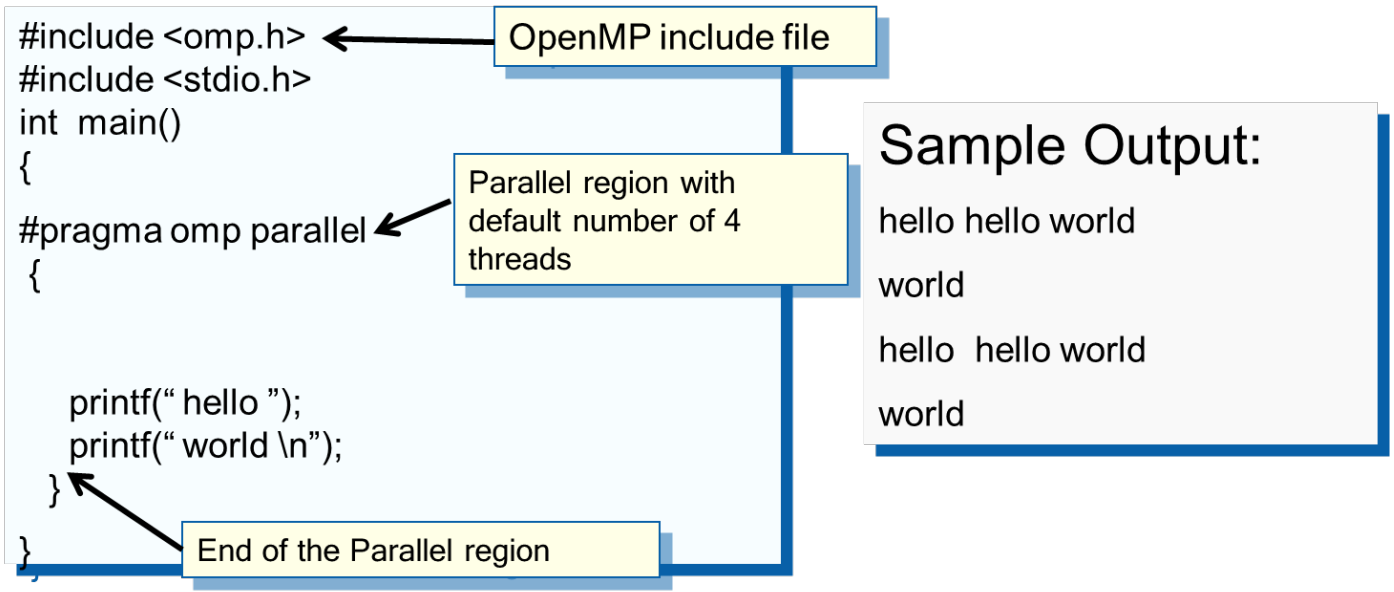
```
#include <omp.h>
#include <stdio.h>
int main()
{
 #pragma omp parallel
 {

    printf(“ hello ”);
    printf(“ world \n”);
 }
}
```

Switches for compiling and linking

gcc –fopenmp	**Gnu (Linux, OSX)**
pgcc -mp pgi	**PGI (Linux)**
icl /Qopenmp	**Intel (windows)**
icc –fopenmp	**Intel (Linux, OSX)**

Example: Hello world in OpenMP

- Write a multithreaded program that prints “hello world”



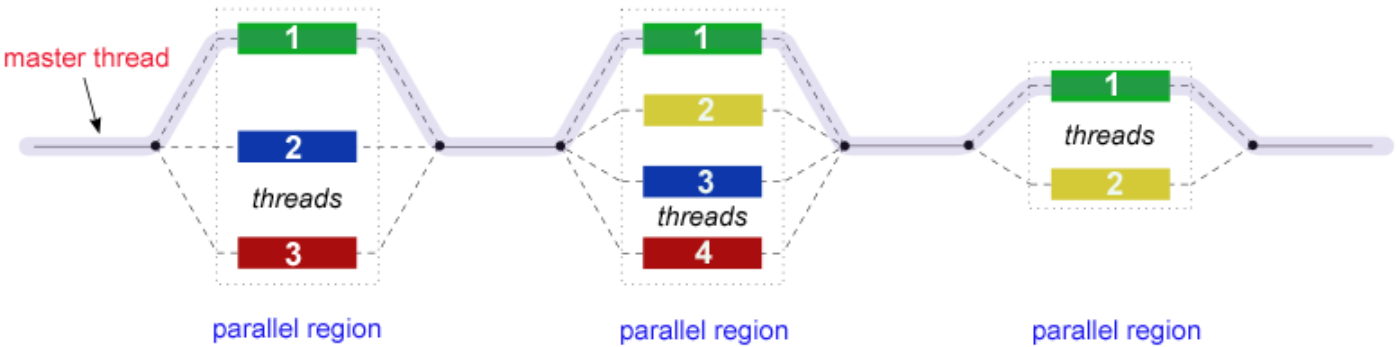
parallel programming

Parallel programming with OpenMP—Creating Threads

OpenMP Programming Model

Fork-Join Parallelism

- All OpenMP programs begin as a single process: the master thread. The master thread executes sequentially until the first parallel region construct is encountered
- Master thread spawns a team of threads as needed
- Parallelism added incrementally until performance goals are met: i.e. the sequential program evolves into a parallel program



Thread creation: Parallel regions example

- Create threads in OpenMP with the parallel construct
  - 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

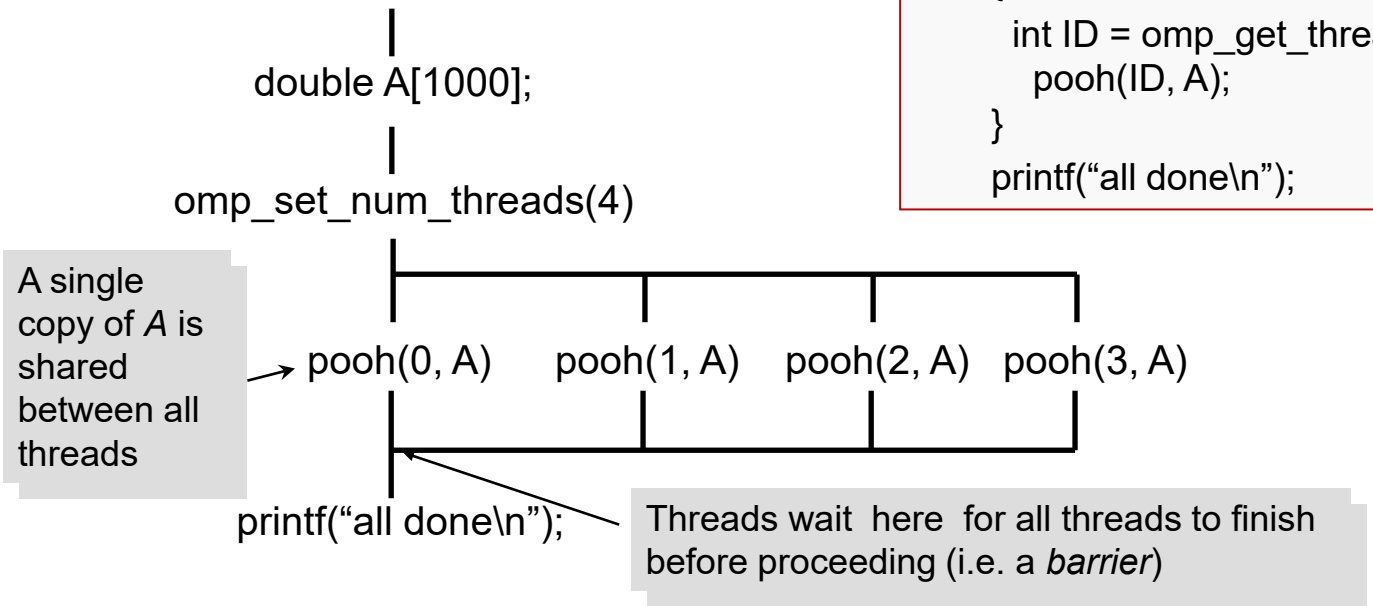
Runtime function returning a thread ID

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

Thread creation: Parallel regions example

- 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");
```



Thread creation: How many threads did you actually get?

- You create a team threads in OpenMP* with the parallel construct
- You can request a number of threads with omp_set_num_threads()
- But is the number of threads requested the number you actually get?
  - NO! An implementation can silently decide to give you a team with fewer threads
  - Once a team of threads is established … the system will not reduce the size of the team

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

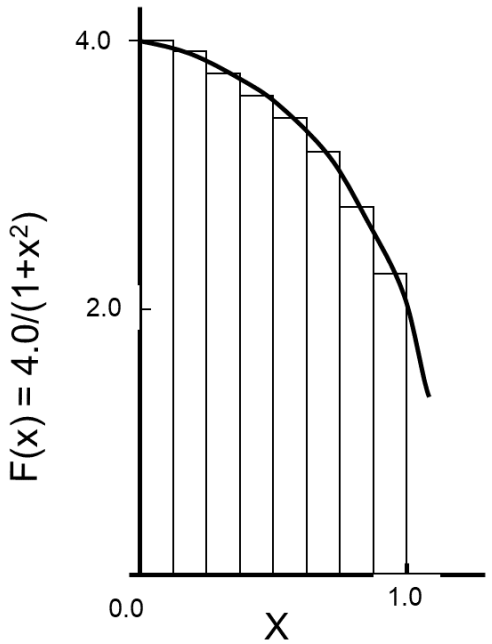
```
double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
        int ID      = omp_get_thread_num();
        int nthrds = omp_get_num_threads();
        pooh(ID,A);
}
```

Runtime function to request a certain number of threads

Runtime function to return actual number of threads in the team

- Each thread calls pooh(ID,A) for `ID` = `0` to `nthrds-1`

An interesting example：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.

An interesting example
Numerical integration: Serial PI program

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

            step = 1.0/(double) num_steps;

            for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
            }
            pi = step * sum;
}
```

An interesting example
Numerical integration:Parallel PI program

- Original Serial pi program with 100000000 steps ran in 1.83 seconds

NUM_THREAD	time
1	1.86
2	1.08
3	0.97
4	0.88

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz

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

parallel programming

Parallel programming with OpenMP—Parallel Loops

The loop worksharing constructs

- Loop worksharing 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);
        }
}
```

Loop construct name

- C/C++: *for*
- Fortran: *do*

The variable *I* is made "private" to each thread by default. You could do this explicitly with a "private(I)" clause

Loop worksharing 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;
        if (id == Nthrds-1)iend = N;
        for(i=istart;i<iend;i++)  { a[i] = a[i] + b[i];}
}
```

OpenMP parallel region and a worksharing for construct

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

Loop worksharing constructs: The schedule clause

- The schedule clause affects how loop iterations are mapped onto threads
  - #pragma omp for **schedule(static [,chunk])**
    - Deal-out blocks of iterations of size "chunk" to each thread
  - #pragma omp for **schedule(dynamic[,chunk])**
    - Each thread grabs "chunk" iterations off a queue until all iterations have been handled

Schedule Clause	When To Use
STATIC	Pre-determined and predictable by the programmer
DYNAMIC	Unpredictable, highly variable work per iteration

Least work at runtime : scheduling done at compile-time

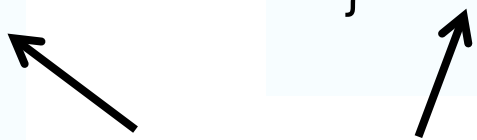
Most work at runtime : complex scheduling logic used at run-time

Combined parallel/worksharing construct

- OpenMP shortcut: Put the "parallel" and the worksharing directive on the same line

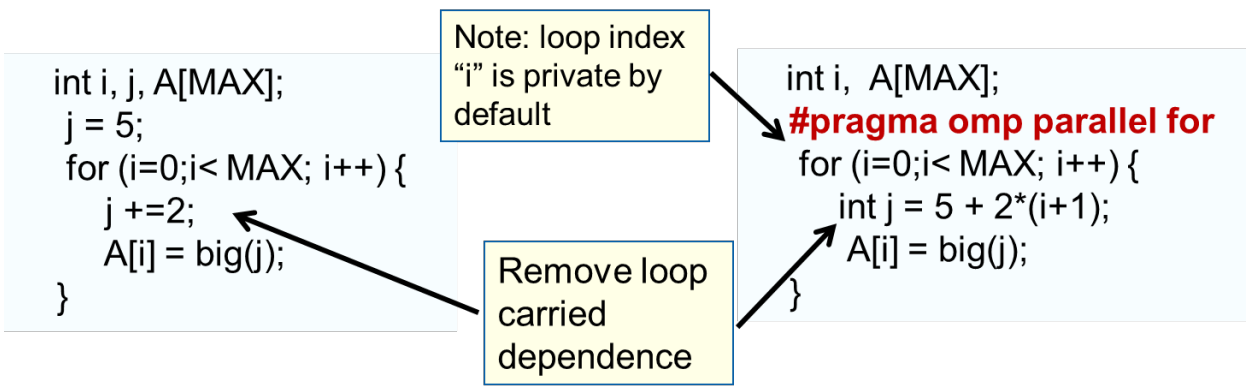
```
double res[MAX]; int i;
#pragma omp parallel
{
    #pragma omp for
    for (i=0;i< MAX; i++) {
        res[i] = huge();
    }
}
```

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



Working with loops

- Basic approach
  - Find compute intensive loops
  - Make the loop iterations independent ... So they can safely execute in any order without loop-carried dependencies
  - Place the appropriate OpenMP directive and test
- Example



Reduction

- **How do we handle this case?**

```
double  ave=0.0, A[MAX];    int i;
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

- We are combining values into a single accumulation variable (ave) … there is a true dependence between loop iterations that can't be trivially removed
- This is a very common situation … it is called a **reduction**
- Support for reduction operations is included in most parallel programming environments

Reduction

- OpenMP *reduction* clause

  reduction (op : list)

- Inside a parallel or a worksharing construct
  - A local copy of each list variable is made and initialized depending on the *op* (e.g. 0 for "+")
  - Updates occur on the local copy
  - Local copies are reduced into a single value and combined with the original global value
- The variables in *list* must be shared in the enclosing parallel region

```
double  ave=0.0, A[MAX];    int i;
#pragma omp parallel for reduction (+:ave)
for (i=0;i< MAX; i++) {
    ave + = A[i];
}
ave = ave/MAX;
```

OpenMP: Reduction Operands/Initial Values

- Many different associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

Operator	Initial value
+	0
*	1
-	0

C/C++ only	
Operator	Initial value
&	~0
|	0
^	0
&&	1
||	0

Fortran Only	
Operator	Initial value
.AND.	.true.
.OR.	.false.
.NEQV.	.false.
.IEOR.	0
.IOR.	0
.IAND.	All bits on
.EQV.	.true.
MIN	Largest pos. number
MAX	Most neg. number

Example: Numerical integration

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{    int i;      double x, pi, sum = 0.0;
     step = 1.0/(double) num_steps;
     #pragma omp parallel
     {
         double x;
        #pragma omp for reduction(+:sum)
                for (i=0;i< num_steps; i++){
                        x = (i+0.5)*step;
                        sum = sum + 4.0/(1.0+x*x);
                }
      }
                pi = step * sum;
}
```

Create a team of threads … without a parallel construct, you'll never have more than one thread

Create a scalar local to each thread to hold value of x at the center of each interval

Break up loop iterations and assign them to threads … setting up a reduction into sum. Note … the loop index is local to a thread by default.

Example: Result of Numerical integration

- Original Serial pi program with 100000000 steps ran in 1.83 seconds

Example: Pi with a loop and a reduction

```
#include <omp.h>
static long num_steps = 100000;        double step;
void main ()
{   int i;          double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    #pragma omp parallel
    {
       double x;
      #pragma omp for reduction(+:sum)
          for (i=0;i< num_steps; i++){
                x = (i+0.5)*step;
                sum = sum + 4.0/(1.0+x*x);
          }
    }
         pi = step * sum;
}
```

NUM_THREAD	time	PI Loop and reduction
1	1.86	1.91
2	1.08	1.02
3	0.97	0.80
4	0.88	0.68

*Intel compiler (icpc) with default optimization level (O2) on Apple OS X 10.7.3 with a dual core (four HW thread) Intel® Core™ i5 processor at 1.7 Ghz and 4 Gbyte DDR3 memory at 1.333 Ghz.

parallel programming

Parallel programming with OpenMP—Synchronization

Synchronization

- High level synchronization
  - critical
  - atomic
  - barrier

Synchronization is used to impose order constraints and to protect access to shared data

Synchronization: critical

- Mutual exclusion: Only one thread at a time can enter a *critical* region

Example:

```
float  res;
#pragma omp parallel
{     float B;    int i, id, nthrds;
      id = omp_get_thread_num();
      nthrds = omp_get_num_threads();
       for(i=id;i<niters;i+=nthrds){
                B =  big_job(i);
#pragma omp critical
            res += consume (B);
      }
}
```

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

Synchronization: atomic

- *atomic* provides mutual exclusion but only applies to the update of a memory location (the update of X in the following example)

Example:

```
#pragma omp parallel
{
        double B, tmp;
        B =  DOIT();
        tmp = big_ugly(B);
 #pragma omp atomic
                X +=  tmp;
}
```

Atomic only protects the read/update of X

Synchronization: barrier

- *barrier*: a point in a program all threads much reach before any threads are allowed to proceed

Example:

```
double Arr[8], Brr[8];          int numthrds;
omp_set_num_threads(8)
#pragma omp parallel
{    int id, nthrds;
     id = omp_get_thread_num();
     nthrds = omp_get_num_threads();
     if (id==0) numthrds = nthrds;
     Arr[id] = big_ugly_calc(id, nthrds);
#pragma omp barrier
     Brr[id] = really_big_and_ugly(id, nthrds, Arr);
}
```

Threads wait until all threads hit the barrier. Then they can go on.

parallel programming

Parallel programming with Open MP—Data Environment

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
  - Both: dynamically allocated memory (ALLOCATE, malloc, new)
- But not everything is shared...
  - Stack variables in subprograms (Fortran) or functions (C) called from parallel regions are PRIVATE
  - Automatic variables within a statement block are PRIVATE

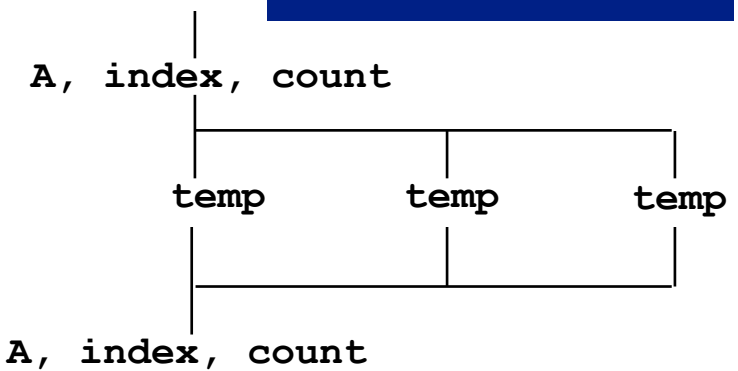
Data Sharing: Examples

```
double A[10];
int main() {
  int index[10];
#pragma omp parallel
    work(index);
  printf("%d\n",
  index[0]);
}
```

```
extern double A[10];
void work(int *index) {
  double temp[10];
  static int count;
  ...
}
```

A, *index* and *count* are shared by all threads

temp is local to each thread



Data Sharing: Changing Storage Attributes

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

All the clauses on this page apply to the OpenMP construct NOT to the entire region

- The final value of a private inside a parallel loop can be transmitted to the shared variable outside the loop with
  - ✓ LASTPRIVATE
- The default attributes can be overridden with
  - ✓ DEFAULT (PRIVATE | SHARED | NONE)

DEFAULT(PRIVATE) *is Fortran only*

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

Data Sharing: Private Clause

- private(*var*) creates a new local copy of *var* for each thread
  - ✓ The value of the private copies is uninitialized
  - ✓ The value of the original variable is unchanged after the region

```
void wrong() {
      int tmp = 0;
#pragma omp parallel for private(tmp)
      for (int j = 0; j < 1000; ++j)
                  tmp += j;
      printf("%d\n", tmp);
}
```

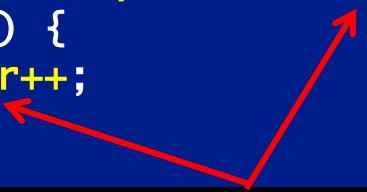
tmp was not initialized

tmp is 0 here

Firstprivate Clause

- Variables initialized from shared variable
- C++ objects are copy-constructed

```
incr = 0;
#pragma omp parallel for firstprivate(incr)
for (i = 0; i <= MAX; i++) {
        if ((i%2)==0) incr++;
        A[i] = incr;
}
```



Each thread gets its own copy of *incr* with an initial value of 0

Lastprivate Clause

- Variables update shared variable using value from last iteration
- C++ objects are updated as if by assignment

```
void sq2(int n, double *lastterm)
{
  double x; int i;
#pragma omp parllel for lastprivate(x)
  for (i = 0; i < n; i++){
     x = a[i]*a[i] + b[i]*b[i];
     b[i] = sqrt(x);
  }
  *lastterm = x;
}
```

x has the value it held for the last sequential iteration (i.e., for i=(n-1))

Data Sharing: Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to use it)
- To change default: DEFAULT(PRIVATE)
  - *each* variable in the construct is made private as if specified in a private clause
  - mostly saves typing

Only the Fortran API supports default(private).

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

Data Sharing: Default Clause

- **default(none)**: **Forces you to define** the storage attributes for variables that appear inside the static extent of the construct … if you fail the compiler will complain. Good programming practice!
- You can put the default clause on parallel and parallel + workshare constructs

The static extent is the code in the compilation unit that contains the construct.

```
#include <omp.h>
int main()
{
    int i, j=5;      double x=1.0, y=42.0;
    #pragma omp parallel for default(none) reduction(*:x)
    for (i=0;i<N;i++){
        for(j=0; j<3; j++)
            x+= foobar(i, j, y);
    }
    printf(" x is %f\n",(float)x);
}
```

The variable *i* is made "private" to each thread by default.

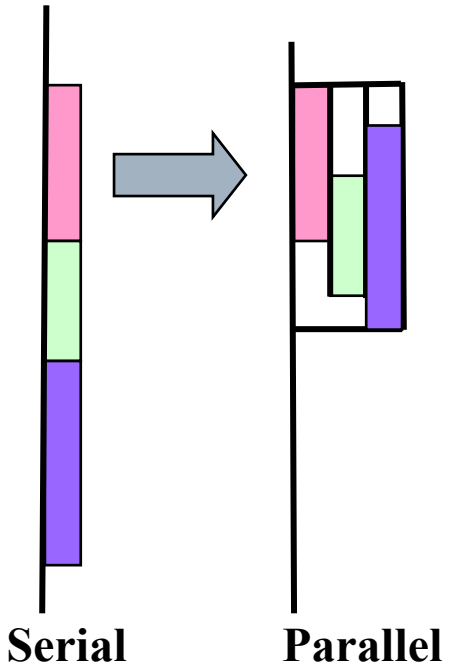
The compiler would complain about j and y, which is important since you don't want j to be shared

parallel programming

Parallel programming with Open MP—Task

What are Tasks?

- Tasks are independent units of work
- Threads are assigned to perform the work of each task
- Tasks may be deferred, may be executed immediately, the runtime system deciding which of the above
- Tasks are composed of
  - code to execute
  - data environment
  - internal control variables (ICV)



Task Construct – Explicit Task View

- A team of threads is created at the omp *parallel* construct
- A single thread is chosen to execute the while loop – lets call this thread "L"
  - A **barrier** is implied at the end of the single block
  - Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the *omp task* construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region's *single* construct

```
#pragma omp parallel
{
  #pragma omp single
  { // block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task private(p)
      process(p);
    p = p->next;  //block 3
    }
  }
}
```

Simple Task Example

```
#pragma omp parallel num_threads(8)
// assume 8 threads
{
 #pragma omp single private(p)
 {
   …
   while (p) {
   #pragma omp task
     {
       processwork(p);
     }
     p = p->next;
   }
 }
}
```

A pool of 8 threads is created here

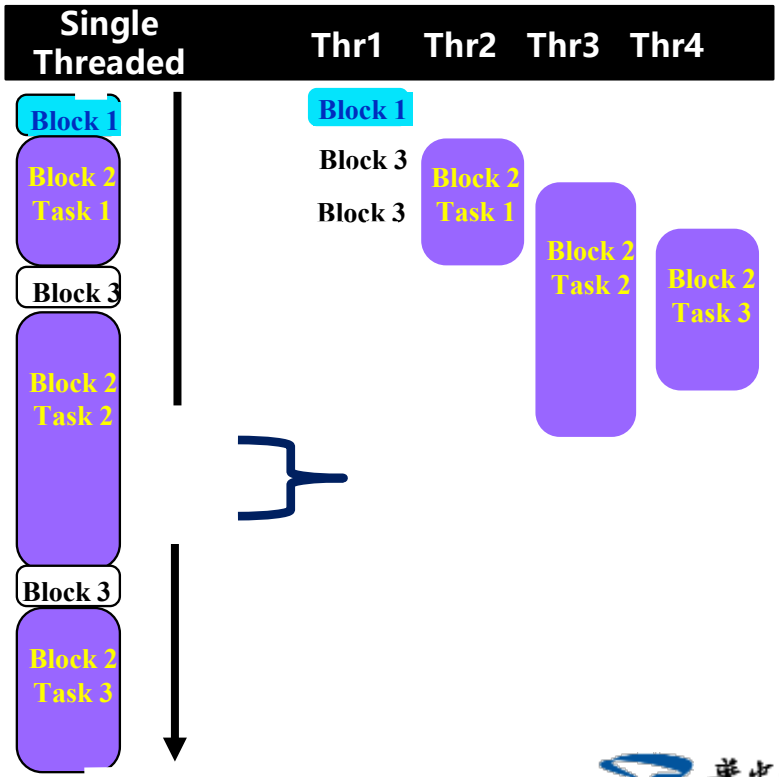
One thread gets to execute the while loop

The single "while loop" thread creates a task for each instance of processwork()

Why are Tasks Useful?

- Have potential to parallelize irregular patterns and recursive function calls

```
#pragma omp parallel
{
  #pragma omp single
  {  // block 1
    node * p = head;
    while (p) {  //block 2
    #pragma omp task
      process(p);
    p = p->next;  //block 3
    }
  }
}
```

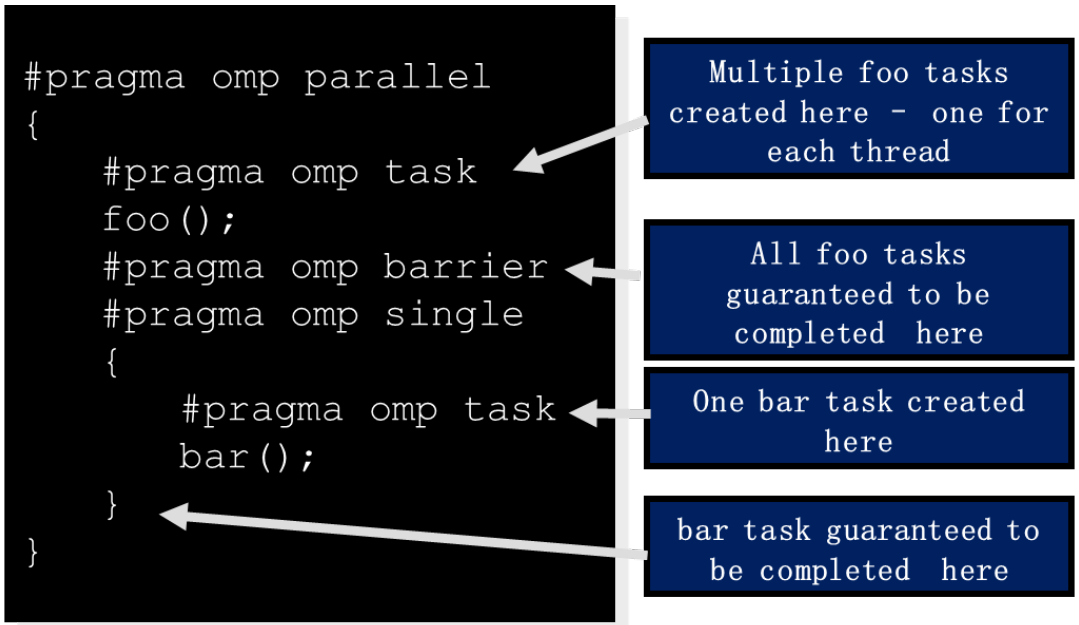


When are Tasks Guaranteed to Complete

- Tasks are guaranteed to be complete at thread barriers

  #pragma omp barrier

Example:



parallel programming

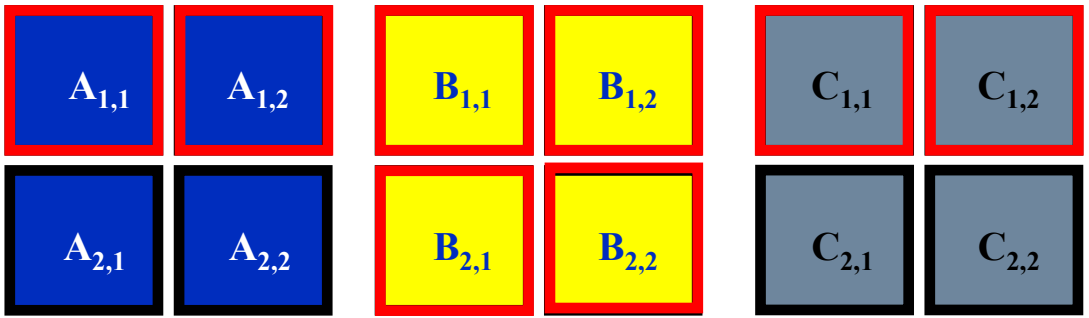
Parallel programming with OpenMP—Recursive Matrix Multiplication

Recursive Matrix Multiplication

- Consider recursive matrix multiplication, described in next 3 slides
  - How would you parallelize this program using OpenMP tasks?
  - What data considerations need to be addressed?

Recursive Matrix Multiplication

- Quarter each input matrix and output matrix
- Treat each submatrix as a single element and multiply
- 8 submatrix multiplications, 4 additions



$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1}$$

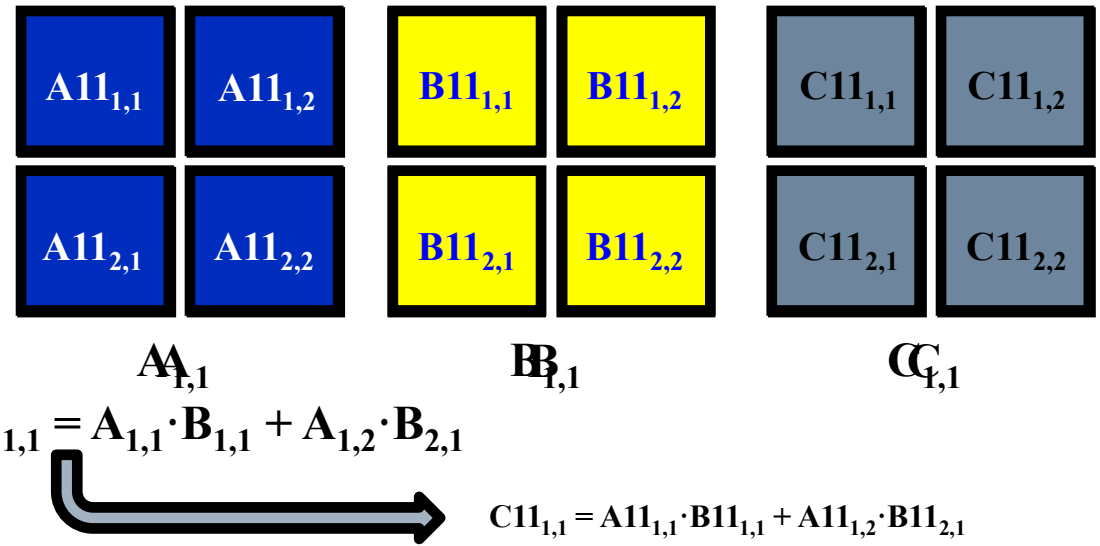
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1}$$

$$C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$

$$C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

How to Multiply Submatrices?

- Use the same routine that is computing the full matrix multiplication
  - Quarter each input submatrix and output submatrix
  - Treat each sub-submatrix as a single element and multiply



Recursively Multiply Submatrices

$$C_{1,1} = A_{1,1} \cdot B_{1,1} + A_{1,2} \cdot B_{2,1} \qquad C_{1,2} = A_{1,1} \cdot B_{1,2} + A_{1,2} \cdot B_{2,2}$$
$$C_{2,1} = A_{2,1} \cdot B_{1,1} + A_{2,2} \cdot B_{2,1} \qquad C_{2,2} = A_{2,1} \cdot B_{1,2} + A_{2,2} \cdot B_{2,2}$$

- **Need range of indices to define each submatrix to be used**

```
void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)
{// Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]   C[mf..ml][nf..nl]

// C11 += A11*B11
     matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);
// C11 += A12*B21
     matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);
   . . .
}
```

- Also need stopping criteria for recursion

Recursive Solution

```
#define THRESHOLD 32768   // product size below which simple matmult code is called

void matmultrec(int mf, int ml, int nf, int nl, int pf, int pl,
                double **A, double **B, double **C)

// Dimensions: A[mf..ml][pf..pl]    B[pf..pl][nf..nl]   C[mf..ml][nf..nl]

{
   if ((ml-mf)*(nl-nf)*(pl-pf) < THRESHOLD)
      matmult (mf, ml, nf, nl, pf, pl, A, B, C);
   else
   {
#pragma omp task
{
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C11 += A11*B11
      matmultrec(mf, mf+(ml-mf)/2, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C11 += A12*B21
}
#pragma omp task
{
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C12 += A11*B12
      matmultrec(mf, mf+(ml-mf)/2, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C12 += A12*B22
}
#pragma omp task
{
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf, pf+(pl-pf)/2, A, B, C);  // C21 += A21*B11
     matmultrec(mf+(ml-mf)/2, ml, nf, nf+(nl-nf)/2, pf+(pl-pf)/2, pl, A, B, C);  // C21 += A22*B21
}
#pragma omp task
{
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf, pf+(pl-pf)/2, A, B, C);  // C22 += A21*B12
     matmultrec(mf+(ml-mf)/2, ml, nf+(nl-nf)/2, nl, pf+(pl-pf)/2, pl, A, B, C);  // C22 += A22*B22
}
#pragma omp taskwait
```

- Could be executed in parallel as 4 tasks
  - Each task executes the two calls for the same output submatrix of C

International Workshop on OpenMP (IWOMP)

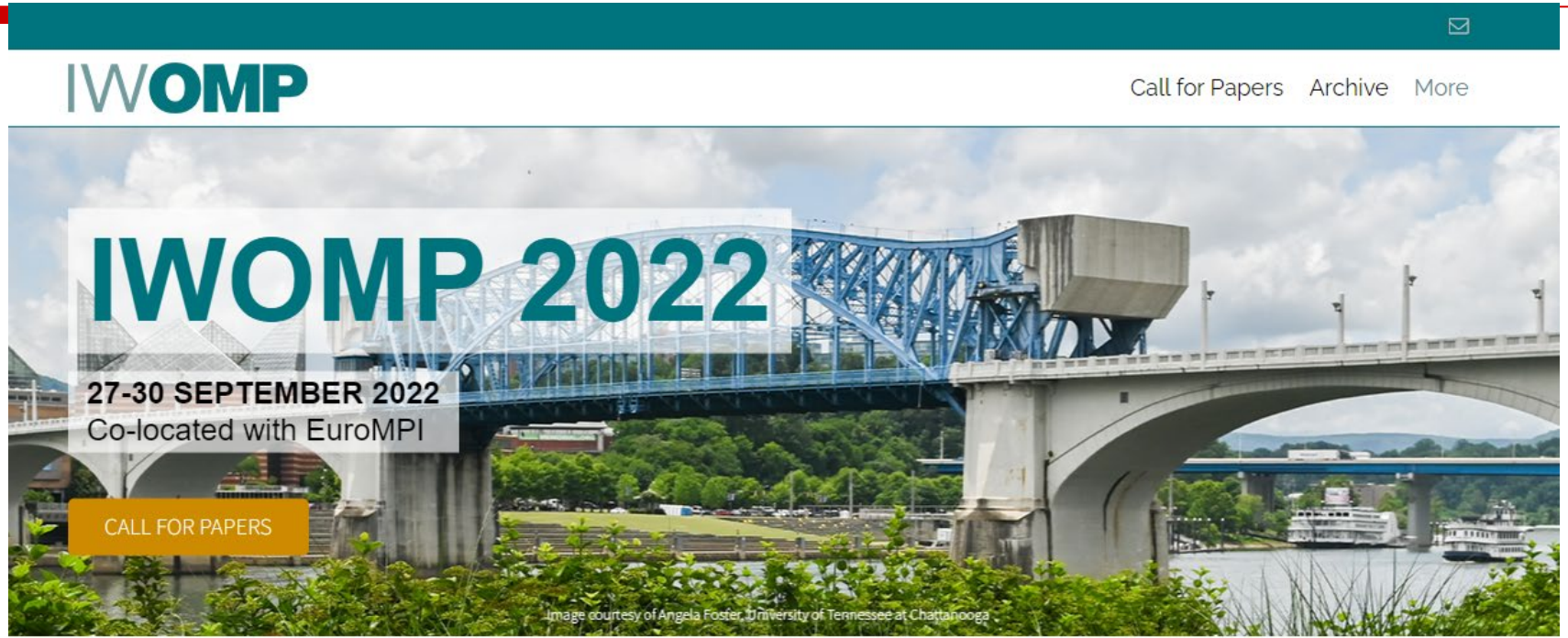


Image courtesy of Angela Foster, University of Tennessee at Chattanooga

18th International Workshop on OpenMP

Conclusion

- A Brief History of Parallel Languages
- OpenMP
  - Shared Memory Programming Model
  - Parallel Programming with Threads
  - OpenMP Overview
  - Parallel programming with OpenMP
    - ✓ Creating Threads
    - ✓ Parallel Loops
    - ✓ Synchronization
    - ✓ Data Environment
    - ✓ Tasks
    - ✓ Example: Recursive Matrix Multiplication