A Tutorial Introduction

Tim Mattson
Intel Corporation
Computational Sciences 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 Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
Parallel Computing
What is it?

- Parallel computing is when a program uses concurrency to either:
  - decrease the runtime for the solution to a problem.
  - Increase the size of the problem that can be solved.

Parallel Computing gives you more performance to throw at your problems.
Parallel Computing:
Writing a parallel application.

Original Problem
Decompose into tasks

Tasks, shared and local data
Group onto execution units.

Code with a parallel Prog. Env.

Units of execution + new shared data for extracted dependencies

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= 0; I<N;I=I+Num){  tmp = func(I);  Res.accumulate( tmp);  }  }

Corresponding source code
Parallel Computing:
The Hardware is in great shape.

<table>
<thead>
<tr>
<th>Time</th>
<th>1998</th>
<th>2000</th>
<th>2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cluster</td>
<td>8 Boxes</td>
<td>32 Boxes</td>
<td>128 Boxes?</td>
</tr>
<tr>
<td>100Mb</td>
<td>VIA</td>
<td>Infiniband</td>
<td></td>
</tr>
<tr>
<td>SMP</td>
<td>1-4 CPUs</td>
<td>1-8 CPUs</td>
<td>1-16 CPUs</td>
</tr>
<tr>
<td>Processor</td>
<td>Pentium® III Xeon™</td>
<td>IA-64 Itanium™</td>
<td>IA-64 McKinley*</td>
</tr>
</tbody>
</table>

*Intel code name
Parallel Computing:
... but where is the software?

- Most ISV’s have ignored parallel computing (other than coarse-grained multithreading for GUI’s and systems programming)

- Why?
  - The perceived difficulties of writing parallel software out-weigh the benefits

- The benefits are clear. To increase the amount of parallel software, we need to reduce the perceived difficulties.
Solution: Effective Standards for parallel programming

- **Thread Libraries**
  - Win32 API
  - POSIX threads.

- **Compiler Directives**
  - OpenMP - portable shared memory parallelism.

- **Message Passing Libraries**
  - MPI - www.mpi-softtech.com
Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
**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: Supporters*

● Hardware vendors
  – Intel, HP, SGI, IBM, SUN, Compaq, Fujitsu

● Software tools vendors
  – KAI, PGI, PSR, APR

● Applications vendors
  – DOE ASCI, ANSYS, Fluent, Oxford Molecular, NAG, Dash, Livermore Software, and many others

*These names of these vendors were taken from the OpenMP web site (www.openmp.org). We have made no attempts to confirm OpenMP support, verify conformity to the specifications, or measure the degree of OpenMP utilization.
OpenMP: 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:
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.

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

Sequential Program

```c
#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: How is OpenMP typically used? (Fortran)

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

Sequential Program

```
program example
double precision Res(1000)
do I=1,1000
call huge_comp(Res(I))
end do
end
```

Parallel Program

```
C$OMP PARALLEL DO
program example
double precision Res(1000)
do I=1,1000
call huge_comp(Res(I))
end do
end
```
OpenMP: 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
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
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 files
  ```
  #include "omp.h"
  ```
OpenMP: Structured blocks

Most OpenMP constructs apply to structured blocks.

- Structured block: a block of code 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++.

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

```c
C$OMP PARALLEL
10    wrk(id) = garbage(id)
30    res(id)=wrk(id)**2
    if(conv(res(id))goto 20
    go to 10
C$OMP END PARALLEL
    if(not_DONE) goto 30
20    print *, id
```

A structured block

Not A structured block
OpenMP: Structured Block Boundaries

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

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

```c
#pragma omp for
for(l=0;l<N;l++){
    res[l] = big_calc(l);
}
```

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

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

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

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
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_thread_num();
    pooh(ID,A);
}
```

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

Each thread calls `pooh(ID)` for ID = 0 to 3
OpenMP: Parallel Regions

- Each thread executes the same code redundantly.

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

- Write a multithreaded program where each thread prints a simple message (such as “hello world”).
- Use two separate printf statements and include the thread ID:
  ```c
  int ID = omp_get_thread_num();
  printf(" hello(\%d ) \n", ID);
  printf(" world(\%d \n", ID);
  ```
- What do the results tell you about I/O with multiple threads?
Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
The “for” Work-Sharing construct splits up loop iterations among the threads in a team.

```c
#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:

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

OpenMP parallel region:

```c
#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:

```c
#pragma omp parallel
#pragma omp for schedule(static)
for(i=0; i<N; i++) { a[i] = a[i] + b[i]; }
```
OpenMP For 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.
OpenMP: Work-Sharing Constructs

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

```c
#pragma omp parallel
#pragma omp sections
{
    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.
OpenMP: Combined Parallel Work-Sharing Constructs

- A short hand notation that combines the Parallel and work-sharing construct.

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

- 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.
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;
}
Agenda

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

Examples of default storage attributes

program sort
common /input/ A(10)
integer index(10)
call instuff
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, count are shared by all threads.

temp is local to each thread
Data Environment: Changing storage attributes

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

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

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

Regardless of initialization, IS is undefined at this point.

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

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.

```fortran
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.

```fortran
program closer
  IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP+ LASTPRIVATE(IS)
  DO J=1,1000
    IS = IS + J
  CONTINUE
1000  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 quick data environment quiz

- Here’s an example of PRIVATE and FIRSTPRIVATE variables A, B, and C = 1
  C$OMP PARALLEL PRIVATE(B)
  C$OMP& FIRSTPRIVATE(C)

- What are A, B and 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

- What are A, B, and C outside this parallel region ...
  - The values of “B” and “C” are undefined.
  - A’s “inside” value is exposed “outside”.
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

These two codes are equivalent

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

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

- Another clause that affects 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 worksharing construct:
  - A local copy of each list variable is made and initialized depending on the “op” (e.g. 0 for “+”)
  - Pair wise “op” is updated on the local value
  - Local copies are reduced into a single global copy at the end of the construct.
OpenMP: Reduction example

```c
#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;
    }
}
```
Exercise 3:  
A multi-threaded “pi” program

- Return to your “pi” program and this time, use private, reduction and a worksharing construct to parallelize it.
- See how similar you can make it to the original sequential program.
Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
OpenMP: Synchronization

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

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

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

```
C$OMP PARALLEL DO PRIVATE(B)
C$OMP& SHARED(RES)
   DO 100 I=1,NITERS
      B = DOIT(I)
C$OMP CRITICAL
      CALL CONSUME (B, RES)
C$OMP END CRITICAL
100 CONTINUE
```
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
C$OMP PARALLEL PRIVATE(B)
    B = DOIT(I)
C$OMP ATOMIC
    X = X + B
C$OMP END PARALLEL
```
**OpenMP: Synchronization**

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

```c
#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 parallel region.
- Implicit barrier at the end of a for work-sharing construct.
- No implicit barrier due to nowait.
OpenMP: Synchronization

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

```c
#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 implied barriers or flushes).

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

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

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

- The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory.
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.

- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

This is a confusing construct and we won’t say much about it. To learn more, consult the OpenMP specifications.
OpenMP: A flush example

- This example shows how flush is used to implement pair-wise synchronization.

```c
integer ISYNC(NUM_THREADS)
C$OMP PARALLEL DEFAULT (PRIVATE) SHARED (ISYNC)
  IAM = OMP_GET_THREAD_NUM()
  ISYNC(IAM) = 0
C$OMP BARRIER
  CALL WORK()
  ISYNC(IAM) = 1  ! I’m all done; signal this to other threads
C$OMP FLUSH(ISYNC)
  DO WHILE (ISYNC(NEIGH) .EQ. 0)
C$OMP FLUSH(ISYNC)
  END DO
C$OMP END PARALLEL
```

Make sure other threads can see my write.
Make sure the read picks up a good copy from memory.
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 critical
  - end single (except when nowait is used)

- Flush is implied on the following OpenMP constructs:
  - barrier
  - critical, end critical
  - end do
  - end parallel
  - end sections
  - end single
  - ordered, end ordered parallel
Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
  - Parallel Regions
  - Worksharing
  - Data Environment
  - Synchronization
  - Runtime functions/environment variables
OpenMP: Library routines

- **Lock routines**
  - `omp_init_lock()`, `omp_set_lock()`, `omp_unset_lock()`, `omp_test_lock()`

- **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()`
  - **Turn on/off nesting and dynamic mode**
    - `omp_set_nested()`, `omp_set_dynamic()`, `omp_get_nested()`, `omp_get_dynamic()`
  - **Are we in a parallel region?**
    - `omp_in_parallel()`
  - **How many processors in the system?**
    - `omp_num_procs()`
OpenMP: Library Routines

- Protect resources with locks.

```c
omp_lock_t lck;
omp_init_lock(&lck);
#pragma omp parallel private (tmp, id)
{
    id = omp_get_thread_num();
tmp = do_lots_of_work(id);
omp_set_lock(&lck);
printf("%d %d", id, tmp);
omp_unset_lock(&lck);
}
```
OpenMP: Library Routines

- To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.

```c
#include <omp.h>
void main()
{
    omp_set_dynamic(0);
    omp_set_num_threads(4);
    #pragma omp parallel
    {
        int id=omp_get_thread_num();
        do_lots_of_stuff(id);
    }
}
```
OpenMP: Environment Variables

- 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`
- Can the program use a different number of threads in each parallel region?
  - `OMP_DYNAMIC TRUE || FALSE`
- Will nested parallel regions create new teams of threads, or will they be serialized?
  - `OMP_NESTED TRUE || FALSE`
Summary: OpenMP Benefits*

- **Get more performance** from applications running on multiprocessor workstations
- **Get software to market sooner** using a simplified programming model
- **Reduce support costs** by developing for multiple platforms with a single source code

Learn more at www.openmp.org

*Disclaimer: these benefits depend upon individual circumstances or system configurations and might not always be available.*
Extra Slides

• Subtle details about OpenMP.
• A Series of numerical integration programs (pi).
• OpenMP references.
• OpenMP command summary to support exercises.
OpenMP: Some subtle details (don’t worry about these at first)

- **Dynamic mode (the default mode):**
  - The number of threads used in a parallel region can vary from one parallel region to another.
  - Setting the number of threads only sets the maximum number of threads - you could get less.

- **Static mode:**
  - The number of threads is fixed between parallel regions.

- **OpenMP lets you nest parallel regions, but…**
  - A compiler can choose to serialize the nested parallel region (i.e. use a team with only one thread).
OpenMP: The if clause

- The if clause is used to turn parallelism on or off in a program:

```c
integer id, N
C$OMP PARALLEL PRIVATE(id) IF(N.gt.1000)
    id = omp_get_thread_num()
    res(id) = big_job(id)
C$OMP END PARALLEL
```

- The parallel region is executed in parallel only if the logical expression in the IF clause is .TRUE.

Make a copy of id for each thread.
OpenMP constructs can span multiple source files.

**poo.f**

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

**bar.f**

```c
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 includes lexical extent*

*Dynamic extent of parallel region includes lexical extent*

*Orphan directives can appear outside a parallel region*
OpenMP: Some subtle details (don’t worry about these at first)

- The data scope clauses take a list argument
  - The list can include a common block name as a short hand notation for listing all the variables in the common block.

- Default private for some loop indices:
  - Fortran: loop indices are private even if they are specified as shared.
  - C: Loop indices on “work-shared loops” are private when they otherwise would be shared.

- Not all privates are undefined
  - Allocatable arrays in Fortran
  - Class type (i.e. non-POD) variables in C++.

See the OpenMP spec. for more details.
OpenMP: More subtle details (don’t worry about these at first)

- Variables privatized in a parallel region cannot be reprivitized on an enclosed `omp for`.
- Assumed size and assumed shape arrays cannot be privatized.
- Fortran pointers or allocatable arrays cannot be `lastprivate` or `firstprivate`.
- When a common block is listed in a data clause, its constituent elements cannot appear in other data clauses.
- If a common block element is privatized, it is no longer associated with the common block.

This restriction will be dropped in OpenMP 2.0
OpenMP: Some subtle details on directive nesting

- For, sections and single directives binding to the same parallel region can’t be nested.
- Critical sections with the same name can’t be nested.
- For, sections, and single can not appear in the dynamic extent of critical, ordered or master.
- Barrier can not appear in the dynamic extent of for, ordered, sections, single., master or critical
- Master can not appear in the dynamic extent of for, sections and single.
- Ordered are not allowed inside critical
- Any directives legal inside a parallel region are also legal outside a parallel region in which case they are treated as part of a team of size one.
Extra Slides

• Subtle details about OpenMP.
• A Series of numerical integration programs (pi).
• OpenMP references.
• OpenMP command summary to support exercises.
PI Program: an example

```c
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**

```c
#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);
}
```

**Doubles code size!**
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];
    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;
}
```

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];
    step = 1.0/(double) num_steps;
    omp_set_num_threads(NUM_THREADS)

#pragma omp parallel
{
    double x;    int id;
    id = omp_get_thread_num();    sum[id] = 0;

#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 ()
{
    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
        }
}

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

```c
#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;
    #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=myrank*my_steps; i<(myrank+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);
}
Extra Slides

- Subtle details about OpenMP.
- A Series of numerical integration programs (pi).
- OpenMP references.
- OpenMP command summary to support exercises.
Reference Material on OpenMP

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

Books on OpenMP: Several books are currently being written on the subject but are not yet available by the time of this writing.

Research papers: There is also an increasing number of papers that discuss experiences, performance, proposed extensions etc. of OpenMP. Two examples of such papers are

- Transparent adaptive parallelism on NOWs using OpenMP; Alex Scherer, Honghui Lu, Thomas Gross, and Willy Zwaenepoel; Proceedings of the 7th ACM SIGPLAN Symposium on Principles and practice of parallel programming, 1999, Pages 96 -106

Extra Slides

• Subtle details about OpenMP.
• A Series of numerical integration programs (pi).
• OpenMP references.
• OpenMP command summary to support exercises.
Some OpenMP commands to use in our labs

- **General usage**
  - `#include “omp.h”`

- **Some subprograms to enable OpenMP**
  - Set and get the number of threads to use
    - `omp_set_num_threads(int NUM_THREADS)`
    - `int nthrds = omp_get_num_threads(void)`
  - Find the thread ID (0 to nthrds-1)
    - `int ID = omp_get_thread_num(void)`
  - Environment variable to set number of threads
    - `OMP_NUM_THREADS int`
More OpenMP for the lab:

- You may want to use the following OpenMP constructs as you do the lab
  - `#pragma omp parallel`
    - creates a parallel region
  - `#pragma omp for`
    - Shares work among the threads in the parallel region.
  - `#pragma omp parallel for`
    - combined parallel region worksharing construct
  - `#pragma omp critical`
    - Creates a critical section for the following block of code.
Even more OpenMP for the lab

- **#pragma omp parallel private(list)**
  - make the comma separated list private to each thread. Can be added to “parallel”, “for”, or “parallel for”

- **#pragma omp parallel for reduction(op:list)**
  - local accumulate into variables in the list, globally reduce each local variable into a single global variable

- **#pragma omp barrier**
  - Each thread waits at the barrier until all threads arrive.

- **#pragma omp master**
  - The following block of code is executed by the master thread only.