Agenda

• Setting the stage
  – Parallel computing, hardware, software, etc.
• OpenMP: A quick overview
• OpenMP: A detailed introduction
Parallel Computing: Writing a parallel application.

Original Problem

Decompose into tasks

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 N = 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);
    }
}
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
Agenda

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

C$OMP THREADPRIVATE(/ABC/)

#pragma omp critical

CALL OMP_SET_NUM_THREADS(10)
call omp_test_lock(jlok)

setenv OMP_SCHEDULE "dynamic"

CALL OMP_INIT_LOCK (ilok)

call OMP_TEST_LOCK (jlok)

C$OMP PARALLEL do shared(a, b, c)

call OMP_INIT_LOCK (ilok)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

C$OMP SINGLE PRIVATE(X)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

C$OMP PARALLEL do ordered private (a, b, c)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

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

#pragma omp parallel for private(A, B)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

C$OMP ORDERED

C$OMP THREADPRIVATE(/ABC/)

#pragma omp critical

CALL OMP_SET_NUM_THREADS(10)
call omp_test_lock(jlok)

setenv OMP_SCHEDULE "dynamic"

CALL OMP_INIT_LOCK (ilok)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

C$OMP PARALLEL do shared(a, b, c)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

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

#pragma omp parallel for private(A, B)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

C$OMP ORDERED

C$OMP THREADPRIVATE(/ABC/)

#pragma omp critical

CALL OMP_SET_NUM_THREADS(10)
call omp_test_lock(jlok)

setenv OMP_SCHEDULE "dynamic"

CALL OMP_INIT_LOCK (ilok)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

C$OMP PARALLEL do shared(a, b, c)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

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

#pragma omp parallel for private(A, B)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

C$OMP ORDERED

C$OMP THREADPRIVATE(/ABC/)

#pragma omp critical

CALL OMP_SET_NUM_THREADS(10)
call omp_test_lock(jlok)

setenv OMP_SCHEDULE "dynamic"

CALL OMP_INIT_LOCK (ilok)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

C$OMP PARALLEL do shared(a, b, c)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

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

#pragma omp parallel for private(A, B)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()

C$OMP ORDERED

C$OMP THREADPRIVATE(/ABC/)

#pragma omp critical

CALL OMP_SET_NUM_THREADS(10)
call omp_test_lock(jlok)

setenv OMP_SCHEDULE "dynamic"

CALL OMP_INIT_LOCK (ilok)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

C$OMP PARALLEL do shared(a, b, c)

C$OMP ATOMIC

setenv OMP_SCHEDULE "dynamic"

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

#pragma omp parallel for private(A, B)

C$OMP PARALLEL COPYIN(/blk/)

C$OMP DO lastprivate(XX)

Nthrds = OMP_GET_NUM_PROCS()
OpenMP Overview:

---

**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

---

Nthrds = OMP_GET_NUM_PROCS()
omp_set_lock(lck)
C$OMP THREADPRIVATE (/ABC/)
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.
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.

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

```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]);
    }
}
```

Sequential Program

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

End User

Application

User layer

Directives
OpenMP library
Environment variables

Prog. Layer (OpenMP API)

Runtime library

System layer

OS/system support for shared memory.
**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:
    ```
    !$OMP construct [clause [clause]...] 
    C$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++.

```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");
```

Not A structured block

```c
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;
goto more;
}
done: if(!really_done()) goto more;
```
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);
}
```

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

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

```fortran
C$OMP PARALLEL DO
    do l=0; l<N; l++)
        res(l) = big_calc(l);
    end do
C$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:

```c
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

  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.

```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)

* 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”.

```c
#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”.

```c
#include "omp.h"
void main()
{
#pragma omp parallel
 {
     int ID = omp_get_thread_num();
     printf(" hello(%d) ", ID);
     printf(" world(%d) 
", 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

```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

```
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];}
```
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

<table>
<thead>
<tr>
<th>Schedule Clause</th>
<th>When To Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATIC</td>
<td>Predictable and similar work per iteration</td>
</tr>
<tr>
<td>DYNAMIC</td>
<td>Unpredictable, highly variable work per iteration</td>
</tr>
<tr>
<td>GUIDED</td>
<td>Special case of dynamic to reduce scheduling overhead</td>
</tr>
</tbody>
</table>

* 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.

```c
#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

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

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

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

These are equivalent
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 \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
PI Program:
The sequential program

```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;
}
```
OpenMP PI Program: Parallel Region example (SPMD Program)

```c
#include <omp.h>
static long num_steps = 100000;
#define NUM_THREADS 2
double step;
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

```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);
}
```
#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
",pi);
}
OpenMP: Scope of OpenMP constructs

OpenMP constructs can span multiple source files.

**poo.f**

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

**Dynamic extent of parallel region includes lexical extent**

**bar.f**

```fortran
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
```

**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**

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

**Subroutine**

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

---

**Data Sharing Examples**

- **A, index, 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

• 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

Regardless of initialization, IS is undefined at this point

```fortran
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.

Program closer

\begin{verbatim}
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
\end{verbatim}

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
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

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

These two codes are equivalent

```c
itotal = 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/. 
Copyprivate

You initialize threadprivate data using a copyprivate clause.

```c
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 initialized
   ... 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:

  \texttt{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.

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

Wednesday, April 15, 2009
OpenMP: Reduction example

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

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

- Here is the correct way to parallelize this code.

```fortran
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.

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>All 1’s</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.OR.</td>
<td>0</td>
</tr>
<tr>
<td>MAX</td>
<td>1</td>
</tr>
<tr>
<td>MIN</td>
<td>0</td>
</tr>
<tr>
<td>//</td>
<td>All 1’s</td>
</tr>
</tbody>
</table>
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

```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;
    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

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

We will save flush for the advanced OpenMP tutorial.
The OpenMP* API

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

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

```c
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()
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.

```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.
- No implicit barrier due to `nowait`.
- Implicit barrier at the end of a for work-sharing construct.
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 synchronization is implied).

```c
#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.

```c
#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
Run time environment routines:

- Modify/Check the number of threads
  - \texttt{omp_set_num_threads()}
  - \texttt{omp_get_num_threads()}
  - \texttt{omp_get_thread_num()}
  - \texttt{omp_get_max_threads()}

- Are we in a parallel region?
  - \texttt{omp_in_parallel()}

- How many processors in the system?
  - \texttt{omp_num_procs()}
OpenMP: Library Routines

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

```c
#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 operation 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_l 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:

Research papers:


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


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.


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.


Extra Slides
A series of parallel pi programs
Some OpenMP Commands to support Exercises
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);
}
```
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: Keep it simple

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

Full control → Increased complexity → 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_thread_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;
}
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=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

```c
#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

```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;
    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;
}
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);
}