OpenMP Overview
openmp.org

Tim Mattson
Intel Corporation
Computational Software Laboratory

Rudolf Eigenmann
Purdue University
School of Electrical and Computer Engineering
OpenMP Release History

- 1997: OpenMP Fortran 1.0
- 1998: OpenMP C/C++ 1.0
- 1999: OpenMP Fortran 1.1
- 2000: OpenMP Fortran 2.0
- 2002: OpenMP C/C++ 2.0
- 2005: OpenMP C/C++/Fortran 2.5
- 2008/9: OpenMP C/C++/Fortran 3.0
- 2011: OpenMP C/C++/Fortran 3.1
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.

Master Thread

Parallel Regions

(usually independent loops or “SPMD” sections)
How is OpenMP typically used?

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

**Split-up this loop between multiple threads**

Program example:

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

Sequential Program

Program example:

```plaintext
C$OMP PARALLEL DO
  do I=1,1000
    call huge_comp(Res(I))
  end do
end
```

Parallel Program
How do threads interact?

- OpenMP is a shared memory model.
  - Threads communicate by sharing variables.

- Unintended sharing of data causes race conditions:
  - race condition: uncoordinated access to shared data; 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.
OpenMP: Some syntax details to get us started

- Most of the constructs in OpenMP are compiler directives or pragmas.
  - For C and C++, the pragmas take the form:
    
    ```
    #pragma omp construct [clause [clause]...] 
    ```
  - For Fortran, the directives take one of the forms:
    
    ```
    C$OMP construct [clause [clause]...] 
    !$OMP construct [clause [clause]...] 
    *$OMP construct [clause [clause]...] 
    ```

- Include file and the OpenMP lib module
  
  ```
  #include <omp.h>
  use omp_lib
  ```
OpenMP: Structured blocks (C/C++)

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

```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 
");
```

```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 more;
  printf(" All done 
");
}
```

A structured block

Not A structured block

* Third party trademarks and names are the property of their respective owner.
OpenMP: Structured blocks (Fortran)

- 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++.

```fortran
C$OMP PARALLEL
10   wrk(id) = garbage(id)
    res(id) = wrk(id)**2
    if(.not.conv(res(id))) goto 10
C$OMP END PARALLEL
    print *, id
```

```fortran
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 {}.

```plaintext
#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$OMP PARALLEL DO
do I=1,N
    res(I)=bigComp(I)
end do
C$OMP END PARALLEL DO
```
OpenMP: Contents

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

- OpenMP is basically the same between Fortran and C/C++
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`
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");
omp_set_num_threads(4)
pooh(1,A)
pooh(2,A)
pooh(3,A)
pooh(0,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.
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 (l=0;l<N;l++){
    NEAT_STUFF(l);
  }
```

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

```
#pragma omp for nowait
```

“nowait” is useful between two consecutive, independent omp for loops.
## Work Sharing Constructs

**A motivating example**

<table>
<thead>
<tr>
<th>Sequential code</th>
<th>OpenMP parallel region</th>
<th>OpenMP parallel region and a work-sharing for-construct</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>for(i=0; i&lt;N; i++) { a[i] = a[i] + b[i]; }</code></td>
<td>```</td>
<td></td>
</tr>
</tbody>
</table>
|                                                                              | ```
|                                                                              | #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];         |
|                                                                              | }                                 |
|                                                                              | #pragma omp parallel            |
|                                                                              | #pragma omp for schedule(static) |
|                                                                              | for(i=0; i<N; i++) { a[i] = a[i] + b[i]; } |
OpenMP For/Do construct:
The schedule clause

- The schedule clause effects how loop iterations are mapped onto threads
  - schedule(static [,chunk])
    - Deal-out blocks of iterations of size “chunk” to each thread.
  - schedule(dynamic[,chunk])
    - Each thread grabs “chunk” iterations off a queue until all iterations have been handled.
  - schedule(guided[,chunk])
    - Threads dynamically grab blocks of iterations. The size of the block starts large and shrinks down to size “chunk” as the calculation proceeds.
  - schedule(runtime)
    - Schedule and chunk size taken from the OMP_SCHEDULE environment variable.
## 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
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.
program sort
common /input/ A(10)
integer index(10)
C$OMP PARALLEL
call work(index)
C$OMP END PARALLEL
print*, index(1)

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

A, index, count
A, index, count
 temp     temp     temp
 temp     temp     temp
 A, index, count

A, index and count are shared by all threads.

temp is local to each thread

* Third party trademarks and names are the property of their respective owner.
Data Environment: Changing storage attributes

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

- 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
  - The original is undefined at the end

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

Regardless of initialization, IS is undefined at this point

IS was not initialized
Firstprivate Clause

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

```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
  1000 CONTINUE
print *, IS
```

- Each thread gets its own IS with an initial value of 0
- IS is defined as its value at the last iteration (i.e. for J=1000)
OpenMP: A data environment test

- Consider this example of PRIVATE and FIRSTPRIVATE

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

- Are A, B, C local to each thread or shared inside the parallel region?
- What are their initial values inside and after the parallel region?

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

Are these two codes equivalent?

\[
\begin{align*}
\text{itotal} &= 1000 \\
\text{C}\$\text{OMP PARALLEL PRIVATE}(np, each) \\
&\quad \text{np} = \text{omp_get_num_threads()} \\
&\quad \text{each} = \text{itotal}/\text{np} \\
&\quad \ldots \\
\text{C}\$\text{OMP END PARALLEL}
\end{align*}
\]

\[
\begin{align*}
\text{itotal} &= 1000 \\
\text{C}\$\text{OMP PARALLEL DEFAULT(PRIVATE) SHARED(itotal)} \\
&\quad \text{np} = \text{omp_get_num_threads()} \\
&\quad \text{each} = \text{itotal}/\text{np} \\
&\quad \ldots \\
\text{C}\$\text{OMP END PARALLEL}
\end{align*}
\]

yes
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.

```fortran
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
```

```fortran
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/.
Copyin

You initialize threadprivate data using a copyin clause.

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

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

C$OMP PARALLEL COPYIN(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:
  
  ```c
  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

```c
#include <omp.h>
void main ()
{
    int i;
    double ZZ, func(), res=0.0;

#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
end program closer
```

● Here is the correct way to parallelize this code.

```fortran
program closer
    IS = 0
    !$omp parallel for reduction(+:IS)
    DO J=1,1000
        IS = IS + J
    1000 CONTINUE
    print *, IS
end program closer
```
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>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
<td>.OR.</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
<td>MAX</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>MIN</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>All 1’s</td>
<td>//</td>
<td>All 1’s</td>
</tr>
</tbody>
</table>
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.
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
```
The OpenMP* API

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

- Only one thread at a time can enter a critical section (with the same name).

```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 <name>
        consum (B, RES);
    }
}
```

Threads wait their turn – only one at a time calls consume()
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$OMP PARALLEL PRIVATE(B)
  B = DOIT(I)
  tmp = big_ugly();
C$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.
- 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 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: Contents

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

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

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

```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 op since Memory stores are not atomic
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`
Agenda

- Setting the stage
  - Parallel computing, hardware, software, etc.
- OpenMP: A quick overview
- OpenMP: A detailed introduction
- OpenMP use in the SPEC OMP Benchmarks
## The SPEC OMPPM2001 Benchmarks

<table>
<thead>
<tr>
<th>Code</th>
<th>Applications</th>
<th>Language</th>
<th>lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>Chemistry/biology</td>
<td>C</td>
<td>13500</td>
</tr>
<tr>
<td>applu</td>
<td>Fluid dynamics/physics</td>
<td>Fortran</td>
<td>4000</td>
</tr>
<tr>
<td>apsi</td>
<td>Air pollution</td>
<td>Fortran</td>
<td>7500</td>
</tr>
<tr>
<td>art</td>
<td>Image Recognition\ neural networks</td>
<td>C</td>
<td>1300</td>
</tr>
<tr>
<td>fma3d</td>
<td>Crash simulation</td>
<td>Fortran</td>
<td>60000</td>
</tr>
<tr>
<td>gafort</td>
<td>Genetic algorithm</td>
<td>Fortran</td>
<td>1500</td>
</tr>
<tr>
<td>galgel</td>
<td>Fluid dynamics</td>
<td>Fortran</td>
<td>15300</td>
</tr>
<tr>
<td>equake</td>
<td>Earthquake modeling</td>
<td>C</td>
<td>1500</td>
</tr>
<tr>
<td>mgrid</td>
<td>Multigrid solver</td>
<td>Fortran</td>
<td>500</td>
</tr>
<tr>
<td>swim</td>
<td>Shallow water modeling</td>
<td>Fortran</td>
<td>400</td>
</tr>
<tr>
<td>wupwise</td>
<td>Quantum chromodynamics</td>
<td>Fortran</td>
<td>2200</td>
</tr>
</tbody>
</table>
## Basic Characteristics

<table>
<thead>
<tr>
<th>Code</th>
<th>Parallel Coverage (%)</th>
<th>Total Runtime (sec)</th>
<th># of parallel sections</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Seq.</td>
<td>4-cpu</td>
</tr>
<tr>
<td>ammp</td>
<td>99.11</td>
<td>16841</td>
<td>5898</td>
</tr>
<tr>
<td>applu</td>
<td>99.99</td>
<td>11712</td>
<td>3677</td>
</tr>
<tr>
<td>apsi</td>
<td>99.84</td>
<td>8969</td>
<td>3311</td>
</tr>
<tr>
<td>art</td>
<td>99.82</td>
<td>28008</td>
<td>7698</td>
</tr>
<tr>
<td>equake</td>
<td>99.15</td>
<td>6953</td>
<td>2806</td>
</tr>
<tr>
<td>fma3d</td>
<td>99.45</td>
<td>14852</td>
<td>6050</td>
</tr>
<tr>
<td>gafort</td>
<td>99.94</td>
<td>19651</td>
<td>7613</td>
</tr>
<tr>
<td>galgel</td>
<td>95.57</td>
<td>4720</td>
<td>3992</td>
</tr>
<tr>
<td>mgrid</td>
<td>99.98</td>
<td>22725</td>
<td>8050</td>
</tr>
<tr>
<td>swim</td>
<td>99.44</td>
<td>12920</td>
<td>7613</td>
</tr>
<tr>
<td>wupwise</td>
<td>99.83</td>
<td>19250</td>
<td>5788</td>
</tr>
</tbody>
</table>

* static sections / sections called at runtime
Wupwise

- Quantum chromodynamics model written in Fortran 90
- Parallelization was relatively straightforward
  - 10 OMP PARALLEL regions
  - PRIVATE and (2) REDUCTION clauses
  - 1 critical section
- Loop coalescing was used to increase the size of parallel sections
CSOMP PARALLEL
CSOMP+ PRIVATE (AUX1, AUX2, AUX3),
CSOMP+ PRIVATE (I, IM, IP, J, JM, JP, K, KM, KP, L, LM, LP),
CSOMP+ SHARED (N1, N2, N3, N4, RESULT, U, X)

CSOMP DO
  DO 100 JKL = 0, N2 * N3 * N4 - 1
    L = MOD (JKL / (N2 * N3), N4) + 1
    LP=MOD(L,N4)+1
    K = MOD (JKL / N2, N3) + 1
    KP=MOD(K,N3)+1
    J = MOD (JKL, N2) + 1
    JP=MOD(J,N2)+1
    DO 100 I=(MOD(J+K+L,2)+1),N1,2
      IP=MOD(I,N1)+1
      CALL GAMMUL(1,0,X(1,(IP+1)/2,J,K,L),AUX1)
      CALL SU3MUL(U(1,1,1,I,J,K,L),'N',AUX1,AUX3)
      CALL GAMMUL(2,0,X(1,(I+1)/2,JP,K,L),AUX1)
      CALL SU3MUL(U(1,1,2,I,J,K,L),'N',AUX1,AUX2)
      CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
      CALL GAMMUL(3,0,X(1,(I+1)/2,J,KP,L),AUX1)
      CALL SU3MUL(U(1,1,3,I,J,K,L),'N',AUX1,AUX2)
      CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
      CALL GAMMUL(4,0,X(1,(I+1)/2,J,K,LP),AUX1)
      CALL SU3MUL(U(1,1,4,I,J,K,L),'N',AUX1,AUX2)
      CALL ZAXPY(12,ONE,AUX2,1,AUX3,1)
      CALL ZCOPY(12,AUX3,1,RESULT(1,(I+1)/2,J,K,L),1)
  100  CONTINUE
CSOMP END DO
CSOMP END PARALLEL
**Swim**

- Shallow Water model written in F77/F90
- Swim is known to be highly parallel
- Code contains several doubly-nested loops
  The outer loops are parallelized

```
$OMP PARALLEL DO
DO 100 J=1,N
DO 100 I=1,M
CU(I+1,J) = .5D0*(P(I+1,J)+P(I,J))*U(I+1,J)
CV(I,J+1) = .5D0*(P(I,J+1)+P(I,J))*V(I,J+1)
Z(I+1,J+1) = (FSDX*(V(I+1,J+1)-V(I,J+1))-FSDY*(U(I+1,J+1)
    -U(I+1,J)))/(P(I,J)+P(I+1,J)+P(I+1,J+1)+P(I,J+1))
H(I,J) = P(I,J)+.25D0*(U(I+1,J)*U(I+1,J)+U(I,J)*U(I,J)
    +V(I,J+1)*V(I,J+1)+V(I,J)*V(I,J))
100 CONTINUE
```
Mgrid

- Multigrid electromagnetism in F77/F90
- Major parallel regions inrprj3, basic multigrid iteration
- Simple loop nest patterns, similar to Swim, several 3-nested loops
- Parallelized through the Polaris automatic parallelizing source-to-source translator
Non-linear PDES time stepping SSOR in F77
Major parallel regions in ssor.f, basic SSOR iteration
Basic parallelization over the outer of 3D loop, temporaries held private

```fortran
!$OMP PARALLEL DEFAULT(SHARED) PRIVATE(M,I,J,K,tmp2)
  tmp2 = dt
!$omp do
  do k = 2, nz - 1
    do j = jst, jend
      do i = ist, iend
        do m = 1, 5
          rsd(m,i,j,k) = tmp2 * rsd(m,i,j,k)
        end do
      end do
    end do
  end do
!$omp end do
!$OMP END PARALLEL
```

Up to 4-nested loops:
Galgel

- CFD in F77/F90
- Major parallel regions in heat transfer calculation
- Loop coalescing applied to increase parallel regions, guided self scheduling in loop with irregular iteration times
!$OMP PARALLEL
!$OMP  DEFAULT(NONE)
!$OMP  PRIVATE (i, IL, J, JL, L, LM, M, LPOP, LPOP1),
!$OMP  SHARED (DX, HtTim, K, N, NKX, NKY, NX, NY, Poj3, Poj4, XP, Y),
!$OMP  SHARED (WXXX, WXXY, WXYX, WXYY, WYXX, WYXY, WYYY, WYYY),
!$OMP  SHARED (WXXX, WXXY, WXYX, WXYY, WYXX, WYXY, WYYY),
If (Ind0 .NE. 1) then
  ! Calculate r.h.s.
  C ++++++ - HtCon(i,j,L)*Z(j)*X(l) ++++++++++++++++++++++++++++++++++++
!$OMP DO SCHEDULE(GUIDED)
Ext12: Do LM = 1, K
  L = (LM - 1) / NKY + 1
  M = LM - (L - 1) * NKY
  Do IL=1,NX
    Do JL=1,NY
      Do i=1,NKX
        Do j=1,NKY
          LPOP( NKY*(i-1)+j, NY*(IL-1)+JL ) =
            WXTX(IL,i,L) * WXY(T(JL,j,M) + WXYT(IL,i,L) * WYTY(JL,j,M)
        End Do
      End Do
    End Do
  End Do
End Do
End Do

C ........... LPOP1(i) = LPOP(i,j)*X(j) .........................
LPOP1(1:K) = MATMUL( LPOP(1:K,1:N), Y(K+1:K+N) )

C ........... Poj3 = LPOP1 ........................................
Poj3( NKY*(L-1)+M, 1:K) = LPOP1(1:K)

C ........... Xp = <LPOP1,Z> ....................................
Xp(NKY*(L-1)+M) = DOT_PRODUCT (Y(1:K), LPOP1(1:K) )

C ........... Poj4(*,i) = LPOP(j,i)*Z(j) .......................
Poj4(NKY*(L-1)+M,1:N) =
  MATMUL ( TRANSPOSE( LPOP(1:K,1:N) ), Y(1:K) )
End Do Ext12
!$OMP END DO

C ............. DX = DX - HtTim*Xp ............................
!$OMP DO
  DO LM = 1, K
    DX(LM) = DX(LM) - DOT_PRODUCT (HtTim(LM,1:K), Xp(1:K))
  END DO
!$OMP END DO NOWAIT

Else

C ************ Jacobian ***************************************
C ........... A = A - HtTim * Poj3 ..........................
!$OMP DO
  DO LM = 1, K
    A(1:K,LM) = A(1:K,LM) -
      MATMUL( HtTim(1:K,1:K), Poj3(1:K,LM) )
  END DO
!$OMP END DO NOWAIT

C ........... A = A - HtTim * Poj4 ..........................
!$OMP DO
  DO LM = 1, N
    A(1:K,LM) = A(1:K,LM) -
      MATMUL( HtTim(1:K,1:K), Poj4(1:K,LM) )
  END DO
!$OMP END DO NOWAIT

End If
!$OMP END PARALLEL
Return
End
APSI

- 3D air pollution model
- Relatively flat profile
- Parts of work arrays used as shared and other parts used as private data

Sample parallel loop from run.f

```fortran
!$OMP PARALLEL
!$OMP+PRIVATE(II,MLAG,HELP1,HELPA1)
!$OMP DO
  DO 20 II=1,NZTOP
    MLAG=NXNY1+II*NXNY
    C
    C      HORIZONTAL DISPERSION PART    2    2    2    2
    C      --- CALCULATE WITH DIFFUSION EIGENVALUES THE K D C/DX,K D C/DY
    C
    CALL DCTDX(NX,NY,NX1,NFILT,C(MLAG),DCDX(MLAG),
                HELP1,HELPA1,FX,FXC,SAVEX)
    IF(NY.GT.1) CALL DCTDY(NX,NY,NY1,NFILT,C(MLAG),DCDY(MLAG),
                             HELP1,HELPA1,FY,FYC,SAVEY)
  20 CONTINUE
!$OMP END DO
!$OMP END PARALLEL
```
Genetic algorithm in C

- Most “interesting” loop: shuffle the population.
  - Original loop is not parallel; performs pair-wise swap of an array element with another, randomly selected element. There are 40,000 elements.
  - Parallelization idea:
    - Perform the swaps in parallel
    - Need to prevent simultaneous access to same array element: use one lock per array element → 40,000 locks.
Parallel loop
In shuffle.f
of Gafort

Exclusive access
to array
elements.
Ordered locking prevents deadlock.

```
!$OMP PARALLEL PRIVATE(rand, iother, itemp, temp, my_cpu_id)
   my_cpu_id = 1
!$   my_cpu_id = omp_get_thread_num() + 1
!$OMP DO
   DO j=1,npopsiz-1
      CALL ran3(1,rand,my_cpu_id,0)
      iother=j+1+DINT(DBLE(npopsiz-j)*rand)
!$      IF (j < iother) THEN
!$         CALL omp_set_lock(lck(j))
!$         CALL omp_set_lock(lck(iother))
!$      ELSE
!$         CALL omp_set_lock(lck(iother))
!$         CALL omp_set_lock(lck(j))
!$      END IF
!$   END DO
!$OMP END DO
!$OMP END PARALLEL
```
Fma3D

- 3D finite element mechanical simulator
- Largest of the SPEC OMP codes: 60,000 lines
- Uses OMP DO, REDUCTION, NOWAIT, CRITICAL
- Key to good scaling was critical section
- Most parallelism from simple DOs
  - Of the 100 subroutines only four have parallel sections; most of them in fma1.f90
- Conversion to OpenMP took substantial work
Parallel loop in platq.f90 of Fma3D

 !$OMP PARALLEL DO &
 !$OMP   DEFAULT(PRIVATE), SHARED(PLATQ,MOTION,MATERIAL,STATE_VARIABLES), &
 !$OMP   SHARED(CONTROL,TIMSIM,NODE,SECTION_2D,TABULATED_FUNCTION,STRESS),&
 !$OMP   SHARED(NUMP4) REDUCTION(+:ERRORCOUNT),                          &
 !$OMP   REDUCTION(MIN:TIME_STEP_MIN),                                   &
 !$OMP   REDUCTION(MAX:TIME_STEP_MAX)

 DO N = 1,NUMP4

 ... (66 lines deleted)

 MatID = PLATQ(N)%PAR%MatID

 CALL PLATQ_MASS ( NEL,SecID,MatID )

 ... (35 lines deleted)

 CALL PLATQ_STRESS_INTEGRATION ( NEL,SecID,MatID )

 ... (34 lines deleted)

 !$OMP END PARALLEL DO

Contains large critical section
SUBROUTINE PLATQ_MASS ( NEL,SecID,MatID )

... (54 lines deleted)

!$OMP CRITICAL (PLATQ_MASS_VALUES)
DO i = 1,4
  NODE(PLATQ(NEL)%PAR%IX(i))%Mass = NODE(PLATQ(NEL)%PAR%IX(i))%Mass + QMass
  MATERIAL(MatID)%Mass = MATERIAL(MatID)%Mass + QMass
  MATERIAL(MatID)%Xcm = MATERIAL(MatID)%Xcm + QMass * Px(I)
  MATERIAL(MatID)%Ycm = MATERIAL(MatID)%Ycm + QMass * Py(I)
  MATERIAL(MatID)%Zcm = MATERIAL(MatID)%Zcm + QMass * Pz(I)
!!
!! Compute inertia tensor B wrt the origin from nodal point masses.
!!
  MATERIAL(MatID)%Bxx = MATERIAL(MatID)%Bxx + (Py(I)*Py(I)+Pz(I)*Pz(I))*QMass
  MATERIAL(MatID)%Byy = MATERIAL(MatID)%Byy + (Px(I)*Px(I)+Pz(I)*Pz(I))*QMass
  MATERIAL(MatID)%Bzz = MATERIAL(MatID)%Bzz + (Px(I)*Px(I)+Py(I)*Py(I))*QMass
  MATERIAL(MatID)%Bxy = MATERIAL(MatID)%Bxy - Px(I)*Py(I)*QMass
  MATERIAL(MatID)%Bxz = MATERIAL(MatID)%Bxz - Px(I)*Pz(I)*QMass
  MATERIAL(MatID)%Byz = MATERIAL(MatID)%Byz - Py(I)*Pz(I)*QMass
ENDDO
!!
!! Compute nodal isotropic inertia
!!
  RMass = QMass * (PLATQ(NEL)%PAR%Area + SECTION_2D(SecID)%Thickness**2) / 12.0D+0
!!
!!
  NODE(PLATQ(NEL)%PAR%IX(5))%Mass = NODE(PLATQ(NEL)%PAR%IX(5))%Mass + RMass
  NODE(PLATQ(NEL)%PAR%IX(6))%Mass = NODE(PLATQ(NEL)%PAR%IX(6))%Mass + RMass
  NODE(PLATQ(NEL)%PAR%IX(7))%Mass = NODE(PLATQ(NEL)%PAR%IX(7))%Mass + RMass
  NODE(PLATQ(NEL)%PAR%IX(8))%Mass = NODE(PLATQ(NEL)%PAR%IX(8))%Mass + RMass
!$OMP END CRITICAL (PLATQ_MASS_VALUES)
!!
!!
RETURN
END
Art

- Image processing
- Good scaling required combining two dimensions into single dimension
- Uses OMP DO, SCHEDULE(DYNAMIC)
- Dynamic schedule needed because of embedded conditional
#pragma omp for private (k,m,n, gPassFlag) schedule(dynamic)
for (ij = 0; ij < ijmx; ij++) {
    j = ((ij/inum) * gStride) + gStartY;
    i = ((ij%inum) * gStride) + gStartX;
    k=0;
    for (m=j;m<(gLheight+j);m++)
    for (n=i;n<(gLwidth+i);n++)
        f1_layer[o][k++].l[0] = cimage[m][n];

    gPassFlag =0;
    gPassFlag = match(o,i,j, &mat_con[ij], busp);

    if (gPassFlag==1) {
        if (set_high[o][0]==TRUE) {
            highx[o][0] = i;
            highy[o][0] = j;
            set_high[o][0] = FALSE;
        }
        if (set_high[o][1]==TRUE) {
            highx[o][1] = i;
            highy[o][1] = j;
            set_high[o][1] = FALSE;
        }
    }
}
}
Molecular Dynamics

- Very large loop in rectmm.c
- Good parallelism required great deal of work
- Uses OMP FOR, SCHEDULE(GUIDED), about 20,000 locks
- Guided scheduling needed because of loop with conditional execution.
```
#pragma omp parallel for private (n27ng0, nng0, ing0, i27ng0, natoms, ii, a1, a1q, a1serial, inclose, ix, iy, iz, inode, nodelistt, r0, r, xt, yt, zt, xt2, yt2, zt2, xt3, yt3, zt3, xt4, yt4, zt4, c1, c2, c3, c4, c5, k, a1VP , a1dpx , a1dpy , a1dpz , a1px, a1py, a1pz, a1qxx , a1qxy , a1qxz ,a1qyy , a1qyz , a1qzz, a1a, a1b, iii, i, a2, j, k1, k2 ,ka2, kb2, v0, v1, v2, v3, kk, atomwho, ia27ng0, iang0, o ) schedule(guided)

for( ii=0; ii< jj; ii++)
...
   for( inode = 0; inode < iii; inode ++)
     if( (*nodelistt)[inode].innode > 0) {
       for(j=0; j< 27; j++)
         if( j == 27  )
           ...
     
     ...
     if( atomwho->serial > a1serial)
           for( kk=0; kk< a1->dontuse; kk++)
             if( atomwho == a1->excluded[kk])

     ...
     for( j=1; j< (*nodelistt)[inode].innode -1 ; j++)

     ...
     if( atomwho->serial > a1serial)
           for( kk=0; kk< a1->dontuse; kk++)
             if( atomwho == a1->excluded[kk]) goto SKIP2;

     ...
     for (i27ng0=0 ; i27ng0<n27ng0; i27ng0++)

     ...
     ...
     for( i=0; i< nng0; i++)
     ...
     if( v3 > mxcut || inclose > NCLOSE )

     ...
```

(Loop body contains 721 lines)
Performance Tuning Example 3: EQUAKE

- **EQUAKE**: Earthquake simulator in C
- EQUAKE is hand-parallelized with relatively few code modifications.
  - Parallelizing the four most time-consuming loops
  - inserted OpenMP pragmas for parallel loops and private data
  - array reduction transformation
  - A change in memory allocation made a big performance difference
```c
/* malloc \texttt{w1[numthreads][ARCHnodes][3]} */

\texttt{#pragma omp parallel for}
for (j = 0; j < numthreads; j++)
  for (i = 0; i < nodes; i++) {
    \texttt{w1[j][i][0] = 0.0; ...;}

\texttt{#pragma omp parallel private(my\_cpu\_id, exp,...)}
{
  \texttt{my\_cpu\_id = omp\_get\_thread\_num();}

\texttt{#pragma omp for}
for (i = 0; i < nodes; i++)
  while (...) {
    ...
    \texttt{exp = loop-local computation;}
    \texttt{w1[my\_cpu\_id][...][1] += exp;}
    ...
  }
}

\texttt{#pragma omp parallel for}
for (j = 0; j < numthreads; j++) {
  for (i = 0; i < nodes; i++) {
    \texttt{w[i][0] += w1[j][i][0]; ...;}
  }
}
### OpenMP Features Used

<table>
<thead>
<tr>
<th>Code</th>
<th>regions</th>
<th>locks</th>
<th>guided</th>
<th>dynamic</th>
<th>critical</th>
<th>nowait</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>7</td>
<td>20k</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>applu</td>
<td>22</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>14</td>
</tr>
<tr>
<td>apsi</td>
<td>24</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>art</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>equake</td>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>fma3d</td>
<td>92/30</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>gafort</td>
<td>6</td>
<td>40k</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>galgel</td>
<td>31/32*</td>
<td></td>
<td>7</td>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td>mgrid</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>11</td>
</tr>
<tr>
<td>swim</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wupwise</td>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

* static sections / sections called at runtime

“Feature” used to deal with NUMA machines: rely on *first-touch* page placement. If necessary, put initialization into a parallel loop to avoid placing all data on the master processor.