Default Clause

- Note that the default storage attribute is DEFAULT(SHARED) (so no need to use it).
- To change default: DEFAULT(PRIVATE)
  - each variable in 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).
OpenMP Reduction

- Combines an accumulation operation across threads:
  
  ```
  reduction (op : list)
  ```

- 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.
  - The variables in “list” must be shared in the enclosing parallel region.

Example from last lecture slightly modified

```c
int var = 0;
#pragma omp parallel for firstprivate (var) lastprivate(var)
for ( j=0; j<1000; j++ ) {
    var = var + j;
}
printf ("%d\n", var);
```

- Parallel code shown here does not lead to the same result as sequential solution
  - Code in last lecture was used to explain demonstrate behavior of firstprivate and lastprivate clauses
Example from last lecture cont.

- E.g. 2 threads, assuming
  - thread 0 has $j = 0$ -> 499
  - thread 1 has $j = 500$ -> 999
- `lastprivate` clause ensure that the variable listed contains the value of the last ‘sequential’ iteration (i.e. $j=1000$)
- In parallel code, $var = \sum_{j=500}^{999} j$
- In the sequential code, $var = \sum_{j=0}^{999} j$

OpenMP reduction solution

```c
var = 0
for ( j=0; j<1000; j++ )
    var = var + j
printf("%d\n", var);
```

- Parallel code leading to the identical result as the sequential version:

```c
var = 0
#pragma omp parallel for reduction(+:var)
for ( j=0; j<1000; j++ )
    var = var + j;
printf("%d\n", var);
```
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>.true.</td>
</tr>
<tr>
<td>.OR.</td>
<td>.false.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>.false.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>.true.</td>
</tr>
<tr>
<td>MIN*</td>
<td>Largest pos. number</td>
</tr>
<tr>
<td>MAX*</td>
<td>Most neg. number</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>iand</td>
<td>All bits on</td>
</tr>
<tr>
<td>ior</td>
<td>0</td>
</tr>
<tr>
<td>ieor</td>
<td>0</td>
</tr>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td>l</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* Min and Max are not available in C/C++

OpenMP: Synchronization

- High level synchronization:
  - critical
  - atomic
  - barrier
  - ordered
- Low level synchronization
  - flush
  - locks (both simple and nested)
Synchronization - critical

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

```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
        consume (B, res);
    }
}
```

Synchronization - atomic

- Atomic provides mutual exclusion execution but only applies to the update of a memory location
  - E.g. the update of x in the following example

```c
#pragma omp parallel private (B)
{
    B = doit(I);
    tmp = big_ugly();
    #pragma omp atomic
    x = x + temp;
}
```
Synchronization - barrier

- **Barrier:** Each thread waits until all threads arrive.
  - Remember: implicit barriers at the end of work-sharing constructs and parallel regions

```c
#pragma omp parallel shared (A,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);
  }
}
```

OpenMP: Synchronization

- The **ordered** region executes in the sequential order.
  - First thread 0, then thread 1, etc...

```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);
  }
```
Parallel Computation
Edgar Gabriel

OpenMP Library Routines

- To use a known, fixed number of threads used in a program,
  - tell the system that you don’t want dynamic adjustment of the number of threads,
  - set the number of threads,
  - check the number of threads you got.
- Note: the system may give you fewer threads than requested. If the precise number of threads matters, test for it and respond accordingly.

```c
int num_threads;
omp_set_dynamic( 0 );
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);
}
```
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

- How to set environment variables
  - Depends on the shell that you use
  - E.g. using bash (default on shark)
    
    export OMP_NUM_THREADS 8

    Or adding export OMP_NUM_THREADS=8 in the .bashrc file in your home directory and logging in freshly

Example: Numerical Integration

Mathematically, we know that:

\[ \int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi \]

We can approximate the integral as a sum of rectangles:

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

Where each rectangle has width \( \Delta x \) and height \( F(x_i) \) at the middle of interval \( i \).
PI Program: an example

static long num_steps = 100000;
double step;
int main ( int argc, char **argv) {
    int i;
    double x, pi, sum = 0.0;
    step = 1.0/(double) num_steps;
    for (i=0; i<= num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0+x*x);
    }
    pi = step * sum;
    return 0;
}

First solution: using parallel regions only

static long num_steps = 100000;
int i, id, nthreads;
double x, pi, sum[NUM_THREADS];
double step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);
#pragma omp parallel private (i, id, x)
{
    id = omp_get_thread_num();
    #pragma omp single
    nthreads = omp_get_num_threads();
    for (i=id, sum[id]=0.0; 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<nthreads; i++)
    pi += sum[i] * step;
Parallel Computation
Edgar Gabriel

Second solution: using critical region

```c
int i, id, nthreads;
double x, pi, sum;
step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);

#pragma omp parallel private (i, id, x, sum)
{
  id = omp_get_thread_num();
  #pragma omp single
  nthreads = omp_get_num_threads();
  for (i=id, sum=0.0;i< num_steps; i=i+nthreads){
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);
  }
  #pragma omp critical
  pi += sum * step;
}
```

Note: this method of combining partial sums doesn’t scale very well.

Third solution: parallel for with a reduction

```c
int i;
double x, pi, sum = 0.0;
step = 1.0/(double) num_steps;
omp_set_num_threads(NUM_THREADS);

#pragma omp parallel for private(x) reduction(+:sum)
for (i=0;i<= num_steps; i++){
  x = (i+0.5)*step;
  sum = sum + 4.0/(1.0+x*x);
}
pi = step * sum;
```

For good OpenMP implementations, reduction is more scalable than critical.

• In practice, you set number of threads by setting the environment variable, OMP_NUM_THREADS

No array, so no false sharing.