Logistics

• Any questions about the lecture or homework?

• Homework 2 done yet??
## Timings for 3(4) functions

### 1000 x 1000 x 1000 case

<table>
<thead>
<tr>
<th>Description</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time to compute C (ddot)</td>
<td>38.28</td>
</tr>
<tr>
<td>Time to compute C (daxpy)</td>
<td>17.16</td>
</tr>
<tr>
<td>Time to compute C (blocked:ddot)</td>
<td>6.95</td>
</tr>
<tr>
<td>Time to compute C (blocked:daxpy)</td>
<td>3.96</td>
</tr>
</tbody>
</table>
OpenMP Programming Model

- The Application Programmer Interface or API is a combination of:
  - Directives
  - Runtime library routines
  - Environment variables
General Directive Sentinels

• **Fortran 77**
  ➢ *$omp directive [options]*
  ➢ C$omp directive [options]
  ➢ !$omp directive [options]

• **Fortran 90,95**
  ➢ !$omp directive [options]

• **C/C++**
  ➢ #pragma omp directive [options]

• **Continuation Syntax:**
  ➢ F77:  !$omp directive
          !$omp+ directive
  ➢ C/C++  #pragma omp directive directive
A little more complex program

```c
#include <stdio.h>
#include <unistd.h>
#include <omp.h>
int main(int argc, char *argv[]) {
  int tid, time_2_sleep;
  printf(" Hello world from threads:\n");
  #pragma omp parallel private(tid)
  {
    tid = omp_get_thread_num();
    printf("<%d>\n",tid);
    time_2_sleep = tid*3 + 5;
    sleep(time_2_sleep);
    printf("--<%d>\n",omp_get_thread_num());
  }
  printf(" I am sequential now\n");
}
```
Output from complex

Hello world from threads:
<0>
<4>
<3>
<1>
<2>
--<0>
--<1>
--<2>
--<3>
--<4>

I am sequential now
Fortran Version more complex

```fortran
program hello2
implicit none
integer tid, time_2_sleep
integer omp_get_thread_num
external omp_get_thread_num
write(6,*)' Hello world from threads:'
!$omp parallel private(tid)
tid = omp_get_thread_num()
time_2_sleep = tid*3 + 5
call sleep(time_2_sleep)
write(6,'(1x,a1,i4,a1)')'<',tid,'>'
!$omp end parallel
write(6,'(1x,a3,i4,a1)')'--<',tid,'>,'

!$omp parallel
write(6,*)' I am sequential now'
end
```
Daxpy Example

#include <stdio.h>
#include <omp.h>
#define VEC_LEN 100
double zp[VEC_LEN], a, y, norm, diff;

int main(int argc, char *argv[]) {
    int i;

double x[VEC_LEN], z[VEC_LEN];
a = (double)1.2;
y = (double)3.2;
daxpy(VEC_LEN, x, z, a, y);
daxpy(VEC_LEN, x, zp, a, y);
norm = (double) 0.0;
for (i=0; i<VEC_LEN; i++) {
    diff = z[i]-zp[i];
    norm += diff*diff;
}
printf(" Norm: %.10e\n", norm);
}
Daxpy Example

```c
void daxpy(int n, double *x, double *z, double a, double y)
{
    int i;
    for(i=0;i<n;i++)
        z[i] = a * x[i] + y;
}

void daxpyp(int n, double *x, double *z, double a, double y)
{
    int i,tid;
    #pragma omp parallel for
    for(i=0;i<n;i++)
        z[i] = a * x[i] + y;
}
```
Note on F77 vs C

• Default variable context rules are different
  ➢ Why, Who knows!!
• When answering questions, say on or about March 2\textsuperscript{nd} you need to conform to the language you are answering with:
  ➢ Q: Construct a vector normalization routine that utilizes OpenMP directives to express the parallelizm.
  ➢ A: …
    ▪ The defaults are different for Fortran and C/C++
    ▪ Be consistent, safety means explicit context scoping of variables involved!
Limits on loops

- **Fortran (no limits 😊)**
  - do i=initial_value,last_value[,stride]

- **C/C++ has limits**
  - for(j=start;j OP end; increment_expression(j))
    - OP can be <, <=,>,>=
      - Note: no ==
    - The increment_expression must modify j
      - j++ or ++j
      - j− or −j
      - j+= increment
      - j-= increment
      - j = j + increment
      - j = j – increment
        - ♠ Note: no * or / !!
Limits on loops [2]

• **C/C++ has limits**
  - for(j=start; j OP end; increment_expression(j))
    - j has to be defined beyond the block context
    - for(int j=start …)
      - Is illegal syntax according to the standard
      - Some compilers handle this some do not!!
  - **Cannot combine indexes**
    - for(j=start, k=0; j<end, k<N; j++, k++)
      - Is illegal syntax
      - Againe some compilers hadle this some do not!!
  - **These issues may be addressed in future versions of the standard but today they are busteded 😊**
Reduction Scope

```c
norm = (double) 0.0;
for (i=0; i<VEC_LEN; i++) {
    diff = z[i] - zp[i];
    norm += diff*diff;
}
```

```c
#pragma omp shared(z, zp) private(i, diff)
 reduction(+:norm)
for (i=0; i<VEC_LEN; i++) {
    diff = z[i] - zp[i];
    norm += diff*diff;
}
```
Reduction Scope

```c
norm = (double) 0.0;
for(i=0;i<VEC_LEN;i++) {
    diff = z[i]-zp[i];
    norm += diff*diff;
}
```

```c
#pragma omp parallel for shared(z,zp)
 private(i,diff) reduction(+:norm)
for(i=0;i<VEC_LEN;i++) {
    diff = z[i]-zp[i];
    norm += diff*diff;
}
```
Reduction Scope

```c
norm = (double) 0.0;
for(i=0;i<VEC_LEN;i++) {
    diff = z[i]-zp[i];
    norm += diff*diff;
}

norm = (double) 0.0;
#pragma omp parallel for shared(z,zp,norm)\ 
    private(i,diff)
for(i=0;i<VEC_LEN;i++) {
    diff = z[i]-zp[i];
#pragma omp critical
    norm += diff*diff;
}
```
Overhead is everything!

- Timings for the three norm examples:
  - Number of Threads: 1
    - Serial Norm: 2.60125286e-11  Time 0.190
    - Region Norm: 1.04050114e-10  Time 0.200
    - for Norm: 2.60125286e-11  Time 0.210
    - Critical Norm: 2.60125286e-11  Time 9.480
  - Number of Threads: 4
    - Serial Norm: 2.60125286e-11  Time 0.190
    - Region Norm: 1.04050114e-10  Time 0.210
    - for Norm: 2.60125286e-11  Time 0.050
    - Critical Norm: 2.60125286e-11  Time 69.280
Overhead is everything!

- **Timings for the three norm examples:**
  - **Number of Threads:** 1
    - **Serial Norm:** 2.60125286e-11  Time 0.190
    - **Region Norm:** 1.04050114e-10 Time 0.200
    - **for Norm:** 2.60125286e-11  Time 0.210
    - **Critical Norm:** 2.60125286e-11 Time 9.480
  
  - **Number of Threads:** 4
    - **Serial Norm:** 2.60125286e-11  Time 0.190
    - **Region Norm:** 1.04050114e-10 Time 0.210
    - **for Norm:** 2.60125286e-11  Time 0.050
    - **Critical Norm:** 2.60125286e-11 Time 69.280
## Timings for the norm examples

<table>
<thead>
<tr>
<th>Number of threads</th>
<th>Region for</th>
<th>Critical Region</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.601e-11</td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td>0.200s</td>
<td>0.210s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9.48s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>2</td>
<td>5.203e-11</td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td>0.200s</td>
<td>0.090s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>46.170s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.00</td>
</tr>
<tr>
<td>3</td>
<td>7.804e-11</td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td>0.200s</td>
<td>0.070s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>57.240s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.00</td>
</tr>
<tr>
<td>4</td>
<td>1.041e-11</td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td>0.200s</td>
<td>0.050s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2.601e-11</td>
</tr>
<tr>
<td></td>
<td></td>
<td>69.280s</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4.00</td>
</tr>
</tbody>
</table>
# Reduction Operators in C

<table>
<thead>
<tr>
<th>Operator</th>
<th>Types</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>int, float, double</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>int, float, double</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>int, float, double</td>
<td>0</td>
</tr>
<tr>
<td>&amp;</td>
<td>int</td>
<td>all bits</td>
</tr>
<tr>
<td></td>
<td></td>
<td>int</td>
</tr>
<tr>
<td>^</td>
<td>int</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>int</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Reduction Operators in Fortran

<table>
<thead>
<tr>
<th>Operator</th>
<th>Types</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>I,FP,C</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>I,FP,C</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>I,FP,C</td>
<td>0</td>
</tr>
<tr>
<td>.AND.</td>
<td>L</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.OR.</td>
<td>L</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>.EQV.</td>
<td>L</td>
<td>.TRUE.</td>
</tr>
<tr>
<td>.NEQV.</td>
<td>L</td>
<td>.FALSE.</td>
</tr>
<tr>
<td>MAX</td>
<td>I,FP</td>
<td>smallest value</td>
</tr>
<tr>
<td>MIN</td>
<td>I,FP</td>
<td>largest value</td>
</tr>
<tr>
<td>IAND</td>
<td>I</td>
<td>all bits set</td>
</tr>
<tr>
<td>IOR</td>
<td>I</td>
<td>0</td>
</tr>
<tr>
<td>IEOR</td>
<td>I</td>
<td>0</td>
</tr>
</tbody>
</table>
Default Variable Scope

• By default all variables are scoped as shared except for:
  - Certain loop index values (loop control values) are made private by default.
  - In subroutine/functions called within a parallel region, local variables and value parameters (C/C++ pass by value arguments) within the subroutine/function are private.
  - Automatic variables (C/C++) declared within the lexical extent of a parallel region are private.
Examples of default scope (F77)

subroutine caller(a,n)
  integer n, a(n), i, j, m
  m=3
$!omp parallel do
  do i = 1, n
    do j = 1, 5
      call callee(a(i), m, j)
    enddo
  enddo
end

subroutine callee(x,y,z)
  common /com/c
  integer x, y, z, c, ii, cnt
  save cnt
  cnt = cnt + 1
  do ii=1, z
    x = y + z
  enddo
end

a shared/(s) declared outside parallel construct
n shared/(s) same
i private/(s) parallel loop index
j private/(s) Fortran sequential loop index
m shared/(s) declared outside parallel construct
x shared/(s) really is a
y shared/(s) really is m
z private/(s) really is j
c shared/(s) in a common block!
ii private/(s) local stack variable in called func.
cnt shared/(u) local variable with save attribute

(s) – safe to use in parallel region
(u) – unsafe to use in parallel region
Examples of default scope (C/C++)

void caller(int *a,int n)
{int i,j,m=3;
#pragma omp parallel for
for(i=0;i<n;i++) {
    int k=m;
    for(j=1;j<=5;j++) {
        callee(&a[i],&k, j);
    }
}
void callee(int *x,int *y,int z)
{ int ii;
    static int cnt;
    cnt++;
    for(ii=1,ii<z;ii++)
        *x = *y + z;
}"}
Scheduling of Loops

• **Static**
  - Choice of which thread performs a particular iteration is solely a function of the iteration number and the number of threads
    - Statically assigned at the beginning of the loop
    - Load imbalance may be an issue if each task has varying work.
    - Lowest or low overhead.

• **Dynamic**
  - The assignment of threads varies at runtime and with the implementation.
    - Each thread gets another set of iterations (a chunk of work) after it completes the current set of work.
    - Load balance of work possible.
    - Higher overhead
Scheduling the loops.

- The default is “static”
- Static can have a chunk size.
- Dynamic with a chunk size
- Guided special dynamic
  - With a minimum chunk size
- **Runtime**
  - Any of the above
  - **MUST** have environment variable set
    - `% setenv OMP_SCHEDULE “static,10”`
- `#pragma omp parallel for schedule(type[,chunk])`
- `!$omp parallel do schedule(type[,chunk])`
Overhead for schedules

- **Static**
  - Lowest overhead
- **Static with a chunk**
  - Low overhead
- **Dynamic (with or without a chunk)**
  - Medium overhead
- **Guided (with or without a chunk)**
  - Highest overhead
- **What does Dynamic mean**
  - Round robin assignment of chunks.
Another example

time0 = Wall_Time();
for(i=0;i<NROWS;i++) {
    for(j=0;j<NCOLS;j++) {
        energy_grid[INDX(i,j,NCOLS)] = energy(i,j);
    }
}
time1 = Wall_Time();

• To parallelize this loop:
  ➢ Energy cannot have any side-effects!!!
  ➢ Cannot modify anything “private” in the loop!
Another example

time0 = Wall_Time();
for(i=0;i<NROWS;i++) {
    for(j=0;j<NCOLS;j++) {
        energy_grid[indx(i,j,NCOLS)] = energy(i,j);
    }
}
time1 = Wall_Time();
time0 = Wall_Time();
#pragma omp parallel for private(i,j)
for(i=0;i<NROWS;i++) {
    for(j=0;j<NCOLS;j++) {
        energy_grid[indx(i,j,NCOLS)] = energy(i,j);
    }
}
time1 = Wall_Time();
printf("Time to generate grid parallel for default(%d) : %.3f\n", nproc, (time1-time0));
Another example [static,chunk]

time0 = Wall_Time();
#pragma omp parallel for private(i,j)\
schedule(static,100)
    for(i=0;i<NROWS;i++) {
        for(j=0;j<NCOLS;j++) {
            energy_grid[INDX(i,j,NCOLS)] =
            energy(i,j);
        }
    }
}
time1 = Wall_Time();
printf("Time to generate grid parallel for\
static,100(%d): %.3f\n",nproc,(time1-time0));
Another example [dynamic]

time0 = Wall_Time();
#pragma omp parallel for private(i,j)
schedule(dynamic)
for(i=0;i<NROWS;i++) {
    for(j=0;j<NCOLS;j++) {
        energy_grid[INDX(i,j,NCOLS)] = energy(i,j);
    }
}
time1 = Wall_Time();
printf("Time to generate grid parallel for
dynamic(%d) : %.3f\n", nproc, (time1-time0));
Another example [guided]

time0 = Wall_Time();
#pragma omp parallel for private(i,j) 
   schedule(guided)
   for(i=0;i<NROWS;i++) {
      for(j=0;j<NCOLS;j++) {
         energy_grid[INDX(i,j,NCOLS)] = energy(i,j);
      }
   }
time1 = Wall_Time();
printf("Time to generate grid parallel for guided(%d) : %.3f\n",nproc,(time1-time0));
Another example [guided,chunk]

time0 = Wall_Time();
#pragma omp parallel for private(i,j) \ schedule(guided,20)
    for(i=0;i<NROWS;i++) {
        for(j=0;j<NCOLS;j++) {
            energy_grid[INDX(i,j,NCOLS)] = energy(i,j);
        }
    }
time1 = Wall_Time();
printf("Time to generate grid parallel for guided,20(%d) : %.3f\n", nproc, (time1-time0));
Time to solution

- default
- static, 100
- dynamic
- guided
- guided, 20

Seconds vs. Threads graph.
Private Variable initialization and finalization

- Normally private variables have an undefined initial value inside the parallel construct!
  - Sometimes we need the private variable value of the variable from the master thread.
  - Sometimes we need to return the private variable value back to the master thread.
    - The sequentially last value (e.g., the value of the last iteration).

- The firstprivate clause initializes EACH thread with the value of a variable from the master thread.
- The lastprivate clause copies the sequentially last value, from the thread that executed that iteration, back to the storage of the master thread.
Caveats of firstprivate and lastprivate

- If a firstprivate variable is initialized only once per thread not once per iteration.
  - If an iteration modifies it other iterations won’t get the updated value.
  - Private means just that!!!

- If the Lastprivate variable is an array or structure then the parallel construct needs to set all values of the array or structure.
  - If it does not then the values of the unassigned array elements or structure components are undefined
    - E.g., garbage 😊