Introduction to OpenMP

Markus Eisenbach
Dmitry Liakh

National Center for Computational Sciences
Shared Memory Multicore: Threads model

- Modern processors contain **multiple** cores
- Each core can execute a sequential code (**thread** of execution)
- All cores in the processor typically have direct access to the same (**shared**) memory pool
- Different cores may have different “cost” of accessing the same memory location (non-uniform memory access, **NUMA**)
- OpenMP is a **programming model** for such multicore shared-memory computer systems
OpenMP controls forking and joining of parallel threads
⇒ parallel regions
and the distribution of work between these threads
⇒ work-sharing constructs
Relaxed Memory Consistency

- Each thread has a PRIVATE view (not copy) of each SHARED memory location
- Possible scenario (timeline):
  Shared Variable A = 42;
  Thread 0 reads A (42);
  Thread 1 writes to A = 24;
  Thread 1 reads A (24);
  Thread 0 reads A (42);
- Synchronization is required to make the memory view consistent (the same) across multiple threads:
  All participating threads must invoke synchronization
What is OpenMP?

OpenMP is NOT a separate programming language!

OpenMP provides a notation to describe to the compiler how the code should be executed in parallel. This is achieved using *directives* that are introduced by:

```
!$omp <directive>            (Fortran)
#pragma omp <directive>       (C and C++)
```

Also a small set of API with functions for retrieving information about OpenMP and setting parameters:

```
#include <omp.h>     use omp_lib
```
Spawning threads: OMP PARALLEL

A code region that to be executed by a team of threads in parallel uses the directive `parallel`

Fortran:

\[
\text{Code outside the parallel region}\\\n\text{!$omp parallel}\\n\text{Code inside the parallel region}\\n\text{!$omp end parallel}\\n\text{Code outside the parallel region}
\]

C / C++:

\[
\text{Code outside the parallel region}\\\n\text{#pragma omp parallel}\\n\text{\{}\\n\text{\hspace{1cm}Code inside the parallel region}\\n\text{\}}\\n\text{Code outside the parallel region}
\]

e.g.:

\[
\text{#pragma omp parallel}\\n\text{printf(“Hello World!\n”);}
\]
Dividing work between threads

How many threads are there?

```c
int omp_get_num_threads();
```

What is my thread index?

```c
int omp_get_thread_num();  
(returns 0 … num_treads-1)
```

e.g.:

```c
#pragma omp parallel
{
    int num = omp_get_num_threads();
    int id = omp_get_thread_num();
    printf("I am thread #\%d of \%d.\n",id,num);
}
```
Valid OpenMP parallel Region

- Branches into and out of a parallel region are illegal
- Do not depend on the ordering of the execution of threads
- Do not rely on the exact number of threads
- In C++ catch/throw must happen in the same thread within a parallel region
- Do not rely on updates to shared memory to become visible to all threads at the same time without explicit synchronisation (The caches might be out of sync.)
**SHARED vs. PRIVATE variables**

- To do useful work we need to share data between threads and have private data that does not interfere with the work in other threads.
- **shared**: each thread accesses the same memory
- **private**: each thread has its own copy
- OpenMP has a default behavior for variables declared outside the parallel region and used inside. (Usually shared). It can be declared with the `default()` clause after `parallel`
- My recommendation: always use `default(none)`
- Explicitly declare the behavior of all variables:
  - `shared(variable names)`
  - `private(variable names)`
  - In C/C++ variables declared within the parallel block are always private.
Example

calculate the square root of each element in a vector of numbers

double a[N], b[N]
...

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

Example

double a[N], b[N]
...
#pragma omp parallel default(none) shared(a,b)
{
    int num_threads = omp_get_num_threads();
    int my_thread = omt_get_thread_num();
    int start = my_thread * N/num_threads;
    int end = start + N/num_threads;
    for(int i=start; i<end; i++)
    {
        b[i] = sqrt(a[i]);
    }
}

There is a bug here. Can you see it?
Example

N might not be divisible by the number of threads

double a[N], b[N]
...
#pragma omp parallel default(none) shared(a,b)
{
    int num_threads = omp_get_num_threads();
    int my_thread = omt_get_thread_num();
    int start = my_thread * (N/num_threads);
    int end = start + N/num_threads;
    if(my_thread == num_threads-1)
        end = N;
    for(int i=start; i<end; i++)
        b[i] = sqrt(a[i]);
}
Work Sharing Constructs

Many parallel execution patterns can be formulated with just the `parallel` directive, yet setting up these patterns becomes tedious and repetitive.

Work-sharing directives implement useful patterns:

- `for / do` parallel loops
- `sections` independently executable code
- `single` executed by a single thread only
- `workshare` Fortran only for implied array loops
- `task` Task based parallelism

(beyond the scope of this tutorial)
For/Do loops are the most common use cases
Instruct OpenMP to distribute a loop over threads:

```c
#pragma omp for (C/C++)
!${omp do} (Fortran)
```

The loop example then can be written as

```c
#pragma omp parallel default(none) shared(a,b)
#pragma omp for
for(int i=0; i<N; i++)
    b[i] = sqrt(a[i]);

!${omp parallel default(none) shared(a,b,N) private(i)
!${omp do
do i=1,N
    b(i) = sqrt(a(i))
end do
!${omp end parallel
```
OMP FOR

Canonical loop form: It must be possible to calculate the iteration count at loop start (no arbitrary while loops!)

```
for(index = start; index comparison end; update)
```

*start* and *end* have to be known at the start of the loop

*comparison* has to be one of <, >, <= or >=

*update* can be

- index++, index--, ++index, --index,
- index += const, index -= const,
- index = index + const, index = index - const
Reduction

calculate the sum of the elements in a vector of numbers

double a[N]
double sum;
...
average = 0.0;
for(int i=0; i<N; i++)
{
    sum += a[i];
}

How can we apply OpenMP to this?
Reduction

This will not work:

double a[N]
double sum;
...
sum = 0.0;
#pragma omp parallel for default(none) shared(sum,a)
for(int i=0; i<N; i++)
{
    sum += a[i];
}

How can we update sum without creating conflicts?
OMP FOR / DO: Reductions

Reductions are operations that update a variable inside of a loop: `var = var op expression`.

Example:

```
sum += a[i];
```

In OpenMP the operation and variable has to be declared in the loop construct with a `reduction` clause

```
reduction(operator : variable)
```

Allowable operators are:

C/C++:

```
+, -, *, &, |, ^, &&, ||
```

Fortran:

```
+, -, *, .and., .or., .eqv., .neqv., max, min, iand, ior, ieor
```
Reduction

C/C++:
sum = 0.0;
#pragma omp parallel for default(none) shared(a) \
    reduction(+:sum)
for(int i=0; i<N; i++)
{
    sum += a[i];
}

Fortran:
sum = 0.0
!$omp parallel do default(none) shared(a,N) private(i) &
!$omp               reduction(+:sum)
do i=1, N
    sum = sum + a(i)
end do
!$omp end parallel do
When there are multiple calculations that are independent, OpenMP can execute them in parallel.

**C/C++**

```c
#pragma omp sections
{
#pragma omp section
    calculationA();
#pragma omp section
    calculationB();
...  
}
```

**Fortran**

```fortran
!$omp sections

!$omp section
    call calculation_A()
!$omp section
    call calculation_B()
...
!$omp end sections
```
OMP CRITICAL

If every thread should be executed, but not at the same time, e.g. when updating a shared variable or memory region, we have to tell OpenMP with the critical construct

```c
#pragma omp critical
{
...
}
```

```c
!$omp critical
...
!$omp end critical
```

This can be used for reduction like cases, which are not covered by the standard form
Example

C/C++:
sum = 0.0;
#pragma omp parallel \
  default(none) shared(a,sum,N)
{
  double local=0.0;
  #pragma omp for
  for(int i=0; i<N; i++)
    local += a[i];

  #pragma omp critical
  sum += local;
}

Fortran:
sum = 0.0
!$omp parallel default(none) &
!$omp shared(a,N) &
!$omp private(i,local)
local = 0.0
!$omp do
doi=1,N
  local = local + a(i)
end do
!$omp critical
sum = sum + local
!$omp end critical
!$omp end parallel
OMP SINGLE / OMP MASTER

We might want to switch to a single thread within a parallel region for executions that are non parallel.

OpenMP provides two constructs for this: `single` and `master`

**C/C++**
```c
#pragma omp single
{
...
}

#pragma omp master
{
...
}
```

**Fortran**
```fortran
!$omp single
...
!$omp end single

!$omp master
...
!$omp end master
!$omp barrier
```
OMP WORKSHARE

Fortran only for Fortran 90 array syntax
Syncing threads: OMP BARRIER
DATA RACE CONDITIONS

- Two or more concurrent operations (by different threads) on the same variable, where at least one operation is WRITE, must be specially protected

- **OMP BARRIER** (synchronization):
  Thread 0 writes to A; Thread 1 writes to B;
  OMP BARRIER (preceding operations completed);
  Thread 0 reads B; Thread 1 reads A;

- **OMP FLUSH** (memory consistency):  
  A=0 (initialized at start);
  Thread 0: A=42;
  Thread 0: OMP FLUSH;
  Thread 1: OMP FLUSH;
  Thread 1: print *,A: 42;

- Some OpenMP directives synchronize implicitly at exit and/or entry
**IMPLICIT SYNCHRONIZATION**

- **Implicit BARRIER:**
  - End of work-sharing directives (DO/FOR, SECTIONS, WORKSHARE), unless NOWAIT clause is specified;
  - End of SINGLE directive, unless NOWAIT is specified;
  - OMP MASTER does not have an implied barrier!

- **Implicit FLUSH:**
  - OMP BARRIER;
  - OMP PARALLEL;
  - OMP CRITICAL;
  - Exit from work-sharing regions, unless NOWAIT;
  - OMP ATOMIC entry/exit with seq_cst;

- **No FLUSH:**
  - Entry to worksharing regions;
  - Entry/exit to/from MASTER;
OpenMP atomic operations are executed atomically (as a whole): Uninterrupted by other atomic ops:

- OMP ATOMIC READ
- OMP ATOMIC WRITE
- OMP ATOMIC UPDATE
- OMP ATOMIC CAPTURE:
  - update then capture;
  - capture then update;
  - capture then write;
Titan Environment Setup

- $MEMBERWORK: Your private space;
- module swap PrgEnv-pgi PrgEnv-gnu;
- git clone https://github.com/DmitryLyakh/OpenMP_tutorial.git
- ftn -fopenmp -O3 main.F90 -lgomp
- gfortran -fopenmp -O3 main.F90 -lgomp
Multithreaded Linear Algebra Library

FORTRAN:
module linear_algebra
    use omp_lib
    implicit none
contains
    subroutine vv_mul(...) 
    end subroutine vv_mul

    subroutine mv_mul(...) 
    end subroutine mv_mul

end module linear_algebra

program main
    use omp_lib
    use linear_algebra
    ... 
end program main

C:
#include <omp.h>
#include <stdio.h>
#include <time.h>

void vv_mul(...){
}

void mv_mul(...){
}
...

int main(int argc, char** argv){
    ...
}
Operation: set vector

FORTRAN:

subroutine v_set(n,v,val)
  integer, intent(in):: n
  real(8), intent(inout):: v(n)
  real(8), intent(in):: val
  integer:: i

  !$OMP PARALLEL DEFAULT(NONE) SHARED(n,v,val) PRIVATE(i)

  !$OMP DO SCHEDULE(STATIC)
    do i=1,n
      v(i)=val
    enddo
  !$OMP END DO

  !$OMP END PARALLEL

end subroutine v_set
Operation: set vector

C:

```c
void v_set(int n, double *v, double val) {
    #pragma omp parallel for default(none) shared(n,v,val)
    for(int i=0; i<n; i++)
        v[i] = val;
}
```
FORTRAN:

subroutine vv_mul(n,v1,v2,d)
  integer, intent(in):: n
  real(8), intent(in):: v1(n),v2(n)
  real(8), intent(out):: d
  integer:: i,nth

  d=0d0

  !$OMP PARALLEL DEFAULT(NONE) SHARED(nth,n,v1,v2) PRIVATE(i) REDUCTION(+:d)

  !$OMP MASTER
    nth=omp_get_num_threads()
  !$OMP END MASTER

  !$OMP DO SCHEDULE(GUIDED)
    do i=1,n
      d=d+v1(i)*v2(i)
    enddo
  !$OMP END DO
  print *,’vv_mul: num_threads = ’,nth

  !$OMP END PARALLEL
end subroutine vv_mul
Operation: dot product

C:

double vv_mul(int n, double *v1, double *v2)
{
    double d;
    int nth;

    d=0.0;

    #pragma omp parallel default(none) shared(nth,n,v1,v2) reduction(+:d)
    {
        #pragma omp master
            nth=omp_get_num_threads();

        #pragma omp for schedule(guided)
            for(int i=0; i<n; i++)
                d += v1[i] * v2[i];
    }
    printf("vv_mul: num_threads = %d\n", nth);

    return d;
}
Operation: vector norm

FORTRAN:

subroutine v_norm2(n,v,d)
    integer, intent(in):: n
    real(8), intent(in):: v(n)
    real(8), intent(out):: d
    integer:: i

    d=0d0

    !$OMP PARALLEL DEFAULT(NONE) SHARED(n,v) PRIVATE(i) REDUCTION(+:d)

    !$OMP DO SCHEDULE(GUIDED)
      do i=1,n
         d=d+v(i)*v(i)
      enddo
    !$OMP END DO

    !$OMP END PARALLEL
    d=sqrt(d)
end subroutine v_norm2
Operation: vector norm

C:

double v_norm2(int n, double *v)
{
    double d;

    d=0.0;

    #pragma omp parallel default(none) shared(n,v) reduction(+:d)
    {
        #pragma omp for schedule(guided)
        for(int i=0; i<n; i++)
            d += v[i] * v[i];
    }

    d = sqrt(d);

    return d;
}
Operation: set matrix

**FORTRAN:**

```fortran
subroutine m_set(n,m,w,val)  
  integer, intent(in):: n,m  
  real(8), intent(inout):: w(n,m)  
  real(8), intent(in):: val  
  integer:: i,j

  !$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w,val) PRIVATE(i,j)

  !$OMP DO SCHEDULE(STATIC)
  do j=1,m
    do i=1,n
      w(i,j)=val
    enddo
  enddo
  !$OMP END DO
  !$OMP END PARALLEL

end subroutine m_set
```
Operation: set matrix

C:

```c
void m_set(int n, int m, double **w, double val)
{
    #pragma omp parallel default(none) shared(n,m,w,val)
    {
        #pragma omp for schedule(static)
        for(int i=0; i<n; i++)
            for(int j=0; j<m; j++)
                w[i][j] = val;
    }
}
```
Operation: set matrix (collapse)

FORTRAN:

subroutine m_set(n,m,w,val)
   integer, intent(in):: n,m
   real(8), intent(inout):: w(n,m)
   real(8), intent(in):: val
   integer:: i,j

!$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w,val) PRIVATE(i,j)

!$OMP DO SCHEDULE(STATIC) COLLAPSE(2)
   do j=1,m
      do i=1,n
         w(i,j)=val
      enddo
   enddo
!$OMP END DO

!$OMP END PARALLEL

end subroutine m_set
C:

```c
void m_set(int n, int m, double **w, double val)
{
    #pragma omp parallel default(none) shared(n,m,w,val)
    {
        #pragma omp for schedule(static) collapse(2)
        for(int i=0; i<n; i++)
            for(int j=0; j<m; j++)
                w[i][j] = val;
    }
}
```
FORTRAN:

subroutine mv_mul(n,m,w1,v1,v2)
integer, intent(in):: n,m
real(8), intent(in):: w1(n,m)
real(8), intent(in):: v1(m)
real(8), intent(inout):: v2(n)
integer:: i,j
!
 !$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w1,v1,v2) PRIVATE(i,j)

 !$OMP DO SCHEDULE(GUIDED)
 do i=1,n
  v2(i)=0d0
  do j=1,m
   v2(i)=v2(i)+w1(i,j)*v1(j)
  enddo
 enddo
 !$OMP END DO

 !$OMP END PARALLEL

 end subroutine mv_mul
Operation: matrix-vector multiplication

C:

```c
void mv_mul(int n, int m, double **w1, double *v1, double *v2)
{
    // w1[n][m], v1[m], v2[n]

    #pragma omp parallel default(none) shared(n,m,w1,v1,v2)
    {
        #pragma omp for schedule(guided)
        for(int i=0; i<n; i++)
        {
            v2[i] = 0.0;
            for(int j=0; j<m; j++)
                v2[i] += w1[i][j] * v1[j];
        }
    }
}
```
Operation: matrix-vector multiplication

FORTRAN:

subroutine mv_mul(n,m,w1,v1,v2)
  integer, intent(in):: n,m
  real(8), intent(in):: w1(n,m)
  real(8), intent(in):: v1(m)
  real(8), intent(inout):: v2(n)
  integer:: i,j

 !$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w1,v1,v2) PRIVATE(i,j)

 !$OMP DO SCHEDULE(GUIDED)
  do i=1,n
    v2(i)=0d0
  enddo
 !$OMP END DO

 !$OMP DO SCHEDULE(GUIDED)  COLLAPSE(2)
  do i=1,n
    do j=1,m
      v2(i)=v2(i)+w1(i,j)*v1(j)
    enddo
  enddo
 !$OMP END DO

 !$OMP END PARALLEL

end subroutine mv_mul
Operation: matrix-vector multiplication

C:

```c
void mv_mul(int n, int m, double **w1, double *v1, double *v2) {
    // w1[n][m], v1[m], v2[n]

    #pragma omp parallel default(none) shared(n, m, w1, v1, v2)
    {
        #pragma omp for schedule(guided)
        for(int i=0; i<n; i++)
            v2[i] = 0.0;

        #pragma omp for schedule(guided) collapse(2)
        for(int j=0; j<m; j++)
            for(int i=0; i<n; i++)
                v2[i] += w1[i][j] * v1[j];
    }
}
```
Operation: matrix-vector multiplication

FORTRAN:

```fortran
subroutine mv_mul(n,m,w1,v1,v2)
  integer, intent(in):: n,m
  real(8), intent(in):: w1(n,m)
  real(8), intent(in):: v1(m)
  real(8), intent(inout):: v2(n)
  integer:: i,j,tmp

  !$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w1,v1,v2) PRIVATE(i,j,tmp)

  !$OMP DO SCHEDULE(GUIDED)
  do i=1,n
    v2(i)=0d0
  enddo
  !$OMP END DO

  !$OMP DO SCHEDULE(GUIDED) COLLAPSE(2)
  do j=1,m
    do i=1,n
      tmp=w1(i,j)*v1(j)
      !$OMP ATOMIC UPDATE
      v2(i)=v2(i)+tmp
    enddo
  enddo
  !$OMP END DO

  !$OMP END PARALLEL

end subroutine mv_mul
```

Presentation_name
operation: matrix-vector multiplication

C:

void mv_mul(int n, int m, double **w1, double *v1, double *v2) {
    // w1[n][m], v1[m], v2[n]

    #pragma omp parallel default(none) shared(n,m,w1,v1,v2)
    {
        #pragma omp for schedule(guided)
        for(int i=0; i<n; i++)
            v2[i] = 0.0;

        #pragma omp for schedule(guided) collapse(2)
        for(int j=0; j<m; j++)
            for(int i=0; i<n; i++)
                {
                    double tmp = w1[i][j] * v1[j];
                    #pragma omp atomic update
                    v2[i] += tmp;
                }
    }
}
Operation: matrix-vector multiplication

FORTRAN:

subroutine mv_mul(n,m,w1,v1,v2)
    integer, intent(in):: n,m
    real(8), intent(in):: w1(n,m)
    real(8), intent(in):  v1(m)
    real(8), intent(inout): v2(n)
    integer:: i,j

!$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w1,v1,v2) PRIVATE(i,j)

!$OMP DO SCHEDULE(GUIDED)
  do i=1,n
    v2 (i)=0d0
  enddo
!$OMP END DO

  do j=1,m
!$OMP DO SCHEDULE(GUIDED)
    do i=1,n
      v2(i)=v2(i)+w1(i,j)*v1(j)
    enddo
!$OMP END DO
  enddo
!$OMP END PARALLEL

end subroutine mv_mul
C:

```c
void mv_mul(int n, int m, double **w1, double *v1, double *v2)
{
    // w1[n][m], v1[m], v2[n]

    #pragma omp parallel default(none) shared(n,m,w1,v1,v2)
    {
        #pragma omp for schedule(guided)
        for(int i=0; i<n; i++)
            v2[i] = 0.0;

        for(int j=0; j<m; j++)
        {
            #pragma omp for schedule(guided)
            for(int i=0; i<n; i++)
                v2[i] += w1[i][j] * v1[j];
        }
    }
}
```
Operation: matrix-vector multiplication

FORTRAN:

```fortran
subroutine mv_mul(n,m,w1,v1,v2)
  integer, intent(in):: n,m
  real(8), intent(in):: w1(n,m)
  real(8), intent(in):: v1(m)
  real(8), intent(inout):: v2(n)
  integer:: i,j

  !$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w1,v1,v2) PRIVATE(i,j)

  !$OMP DO SCHEDULE(GUIDED)
  do i=1,n
    v2(i)=0d0
  enddo
  !$OMP END DO

  do j=1,m
    !$OMP DO SCHEDULE(GUIDED)
    do i=1,n
      v2(i)=v2(i)+w1(i,j)*v1(j)
    enddo
    !$OMP END DO  NOWAIT
  enddo
  !$OMP END PARALLEL

  end subroutine mv_mul
```
C:

void mv_mul(int n, int m, double **w1, double *v1, double *v2)
{
    // w1[n][m], v1[m], v2[n]

    #pragma omp parallel default(none) shared(n,m,w1,v1,v2)
    {
        #pragma omp for schedule(guided)
        for(int i=0; i<n; i++)
            v2[i] = 0.0;

        for(int j=0; j<m; j++)
        {
            #pragma omp for schedule(guided) nowait
            for(int i=0; i<n; i++)
                v2[i] += w1[i][j] * v1[j];
        }
    }
}
Operation: matrix norm

FORTRAN:

subroutine m_norm2(n,m,w,d)
  integer, intent(in):: n,m
  real(8), intent(in): w(n,m)
  real(8), intent(out):: d
  integer:: i,j

  d=0d0
  !$OMP PARALLEL DEFAULT(NONE) SHARED(n,m,w) PRIVATE(i,j) REDUCTION(+:d)
  !$OMP DO SCHEDULE(GUIDED) COLLAPSE(2)
  do j=1,m
    do i=1,n
      d=d+w(i,j)*w(i,j)
    enddo
  enddo
  !$OMP END DO
  !$OMP END PARALLEL
  d=sqrt(d)
end subroutine m_norm2
Operation: matrix norm

C:

double m_norm2(int n, int m, double **w)
{
    // w[n][m]

double d=0.0;

#pragma omp parallel default(none) shared(n,m,w) reduction(+:d)
{
    #pragma omp for schedule(guided) collapse(2)
    for(int i=0; i<n; i++)
        for(int j=0; j<m; j++)
            d += w[i][j] * w[i][j];
}
d = sqrt(d);
return d;
}
-operation: matrix-matrix multiplication

FORTRAN:

```fortran
subroutine mm_mul(n,m,l,w1,w2,w3)
  integer, intent(in):: n, m, l
  real(8), intent(in):: w1(n, l), w2(l, m)
  real(8), intent(inout):: w3(n, m)
  integer:: i, j, k
  real(8):: tmp

  !$OMP PARALLEL DEFAULT(NONE) SHARED(n, m, l, w1, w2, w3) PRIVATE(i, j, k, tmp)
  !$OMP DO SCHEDULE(GUIDED) COLLAPSE(2)
  do i=1, n
    do j=1, m
      tmp=0d0
      do k=1, l
        tmp=tmp+w1(i, k)*w2(k, j)
      enddo
      w3(i, j)=tmp
    enddo
  enddo
  !$OMP END DO
  !$OMP END PARALLEL

end subroutine mm_mul
```

Operation: matrix-matrix multiplication

C:

```c
void mm_mul(int n, int m, int l, const double **w1, const double **w2, double **w3)
{
    // w1[n][l], w2[l][m], w3[n][m]

    #pragma omp parallel default(none) shared(n,m,l,w1,w2,w3)
    {
        #pragma omp for schedule(guided) collapse(2)
        for(int i=0; i<n; i++)
            for(int j=0; j<m; j++)
            {
                double tmp = 0.0;
                for(int k=0; k<l; k++)
                    tmp += w1[i][k] * w2[k][j];
                w3[i][j] = tmp;
            }
    }
}
```
Titan Environment Setup

- $MEMBERWORK: Your private space;
- module swap PrgEnv-pgi PrgEnv-gnu;
- git clone https://github.com/DmitryLyakh/OpenMP_tutorial.git
- ftn -fopenmp -O3 main.F90 -lgomp
- gfortran -fopenmp -O3 main.F90 -lgomp
- omp_set_num_threads(1): Check correctness!
- omp_set_num_threads(>1): Check correctness!
- omp_set_num_threads(>1): Time it: Scalability benchmark
- Play with the matrix-matrix multiplication to make it as fast as possible (on Titan)!
- Fortran: mm_flops() function measures GFlop/s
- C timing: double omp_get_wtime(): Time in seconds
Optimizations

- Always inspect the memory access pattern for each case: Read contiguously as much as possible!
- Avoid races: Correctness!
- Avoid false sharing: Multiple threads accessing disjoint parts of a cache line, with at least one thread writing to it.
- Reorder loops if possible and beneficial for performance.
- Examine expected loop ranges. Collapse loops if necessary.
- Specify explicit loop iteration scheduling in OpenMP.
- Loop range blocking for cache friendly execution.