Lecture 6

OpenMP: Synchronization

- The ordered construct enforces the sequential order for a block.

```
#pragma omp parallel private (tmp)
#pragma omp for ordered
        for (I=0;I<N;I++){
                tmp = NEAT_STUFF(I);
#pragma ordered
                res = consum(tmp);
        }
```

Ordered Sections

- Impose an order across the iterations of a parallel loop
- Identify a portion of code within each loop iteration that must be executed in the original, sequential order of the loop iterations.
- Restrictions:

  - If a parallel loop contains an ordered directive, then the parallel loop directive itself must contain the ordered clause.

  - An iteration of a parallel loop is allowed to encounter at most one ordered section.

```
#pragma omp parallel for ordered
        for (i=0; i<n; i++)
            {
             a[i] = … complex calculation here …

// Wait until the previous iteration has finished its section
#pragma omp ordered
             print ("%f\n", a[i]);
            }
```

OpenMP: Synchronization

- The **master** construct denotes a structured block that is only executed by the master thread. The other threads just skip it (no implied barriers or flushes).

```
#pragma omp parallel private (tmp)
{
        do_many_things();
#pragma omp master
        {    exchange_boundaries();   }
#pragma barrier
        do_many_other_things();
}
```

OpenMP: Synchronization

- The **flush** construct denotes a sequence point where a thread tries to create a consistent view of memory.
  - All memory operations (both reads and writes) defined prior to the sequence point must complete.
  - All memory operations (both reads and writes) defined after the sequence point must follow the flush.
  - Variables in registers or write buffers must be updated in memory.
- Arguments to flush specify which variables are flushed. No arguments specifies that all thread visible variables are flushed.

OpenMP:
A flush example

- This example shows how **flush** is used to implement pair-wise synchronization.

```
int isync[2];  // Two threads
#pragma omp parallel shared(isync)
{
 int ID = omp_get_thread_num();
 isync[ID] = 0;
 #pragma omp barrier
 main_work();
 isync[ID] = 1; // I'm all done; signal this to the other thread
 #pragma omp flush(isync)
 while (isync[1-ID]==0) {
    side_work();
    #pragma omp flush(isync)
    }
}
```

Note – the following examples illustrate the ordering properties of the flush operation. In the following incorrect pseudocode example, the programmer intends to prevent simultaneous execution of the critical section by the two threads, but the program does not work properly because it does not enforce the proper ordering of the operations on variables **a** and **b**.

Incorrect example:

```
a = b = 0
```

thread 1	*thread 2*
`b = 1`	`a = 1`
flush `(b)`	*flush* `(a)`
flush `(a)`	*flush* `(b)`
`if (a == 0) then`	`if (b == 0) then`
critical section	*critical section*
`end if`	`end if`

The problem with this example is that operations on variables **a** and **b** are not ordered with respect to each other.

For instance, nothing prevents the compiler from moving the flush of **b** on thread 1 or the flush of **a** on thread 2 to a position completely after the critical section (assuming that the critical section on thread 1 does not reference **b** and the critical section on thread 2 does not reference **a**).

If either re-ordering happens, the critical section can be active on both threads simultaneously.

The following correct pseudocode example correctly ensures that the critical section is executed by not more than one of the two threads at any one time. Notice that execution of the critical section by neither thread is considered correct in this example.

Correct example:

```
a = b = 0
```

thread 1	*thread 2*
`b = 1`	`a = 1`
flush `(a,b)`	*flush* `(a,b)`
`if (a == 0) then`	`if (b == 0) then`
critical section	*critical section*
`end if`	`end if`

The compiler is prohibited from moving the flush at all for either thread, ensuring that the respective assignment is complete and the data is flushed before the `if` statement is executed.

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 critical
end single (except when nowait is used)
```

- Flush is implied on the following OpenMP constructs:

```
barrier
critical, end critical
end do
end parallel
```

```
end sections
end single
ordered, end ordered
```

OpenMP: Some subtle details on directive nesting

- *For*, *sections* and *single* directives binding to the same parallel region can't be nested.
- Critical sections with the same name can't be nested.
- *For*, *sections*, and *single* can not appear in the dynamic extent of *critical*, *ordered* or *master*.
- *Barrier* can not appear in the dynamic extent of *for*, *ordered*, *sections*, *single*., *master* or *critical*
- *Master* can not appear in the dynamic extent of *for*, *sections* and *single*.
- *Ordered* are not allowed inside *critical*
- Any directives legal inside a parallel region are also legal outside a parallel region in which case they are treated as part of a team of size one.

OpenMP: Contents

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

OpenMP: Library routines

- Lock routines
  - omp_init_lock(), omp_set_lock(), omp_unset_lock(), omp_test_lock()
- 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()
  - Turn on/off nesting and dynamic mode
    - omp_set_nested(), omp_set_dynamic(), omp_get_nested(), omp_get_dynamic()
  - Are we in a parallel region?
    - omp_in_parallel()
  - How many processors in the system?
    - omp_num_procs()

Lock: low-level synchronization functions

- **When to use lock**

  1) When the synchronization protocols required by a problem cannot be expressed with OpenMP's high-level synchronization constructs.

  2) When the parallel overhead incurred by OpenMP's high-level synchronization constructs is too large.

The simple lock routines are as follows:

- **omp_init_lock** routine initializes a simple lock.
- **omp_destroy_lock** routine destroys a simple lock.
- **omp_set_lock** routine waits until a simple lock is available, and then sets it.
- **omp_unset_lock** routine unsets a simple lock.
- **omp_test_lock** routine tests a simple lock, and sets it if it is available.

C/C++	Fortran
void omp_init_lock(omp_lock_t **lock*);	integer (kind=omp_lock_kind) *svar* subroutine omp_init_lock(*svar*)

OpenMP: Library Routines

- Protect resources with locks.

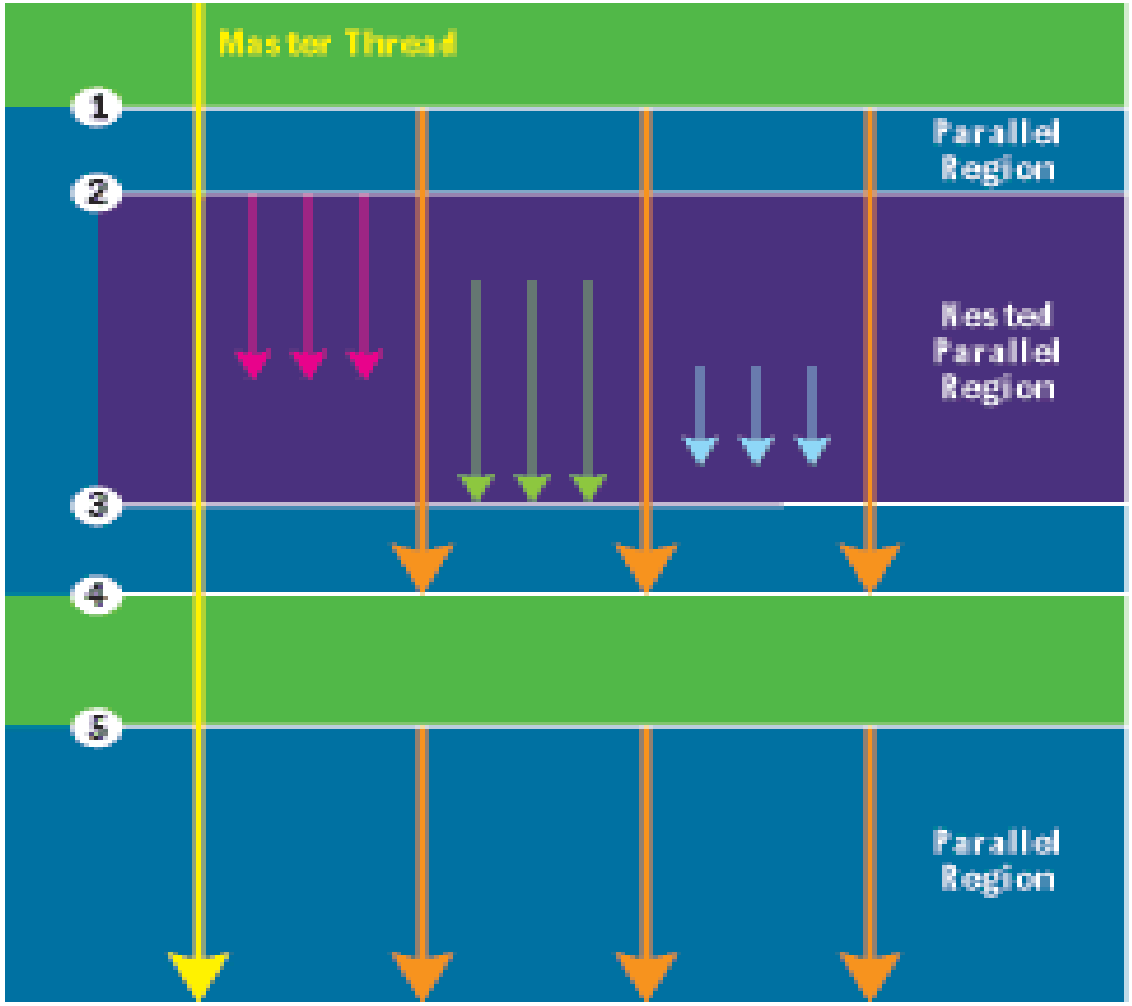
```
    omp_lock_t lck;
    omp_init_lock(&lck);
#pragma omp parallel private (id, tmp)
{
    id = omp_get_thread_num();
    tmp = do_lots_of_work(id);
    omp_set_lock(&lck);
    printf("%d %d", id, tmp);
    omp_unset_lock(&lck);
}
```

OpenMP: Library Routines

- To fix the number of threads used in a program, first turn off dynamic mode and then set the number of threads.

```
#include <omp.h>
void main()
{   omp_set_dynamic(0);
    omp_set_num_threads(4);
#pragma omp parallel
    {   int id=omp_get_thread_num();
        do_lots_of_stuff(id);   }
}
```

OpenMP excution model (nested parallel)



```
program LIB_ENV
    use omp_lib
    implicit none
    integer :: nthreads
    logical :: dynamics, nnested
    integer :: myid
    write(*,*) "start"
    nthreads = omp_get_num_threads()
    dynamics = omp_get_dynamic()
    nnested = omp_get_nested()
    write(*,*) "nthreads, dynamics, nnested : ", nthreads, dynamics, nnested
    write(*,*) "before"
    !$omp parallel private(myid)
        !$omp master
        nthreads = omp_get_num_threads()
        dynamics = omp_get_dynamic()
        nnested = omp_get_nested()
        write(*,*) "nthreads, dynamics, nnested : ", nthreads, dynamics, nnested
        !$omp end master
        myid = omp_get_thread_num()
        write(*,*) "myid : ", myid
    !$omp end parallel
    write(*,*) "after"
end program
```

/home/syam/ces745/openmp/Fortran/data-scope

```
[~] ifort -qopenmp -o openmp_lib_env-f90 openmp_lib_env.f90
[~] ./openmp_lib_env
 start
 nthreads, dynamics, nnested :          1 F F
 before
 nthreads, dynamics, nnested :          8 F F
 myid :          0
 myid :          3
 myid :          2
 myid :          1
 myid :          4
 myid :          7
 myid :          6
 myid :          5
 after
```

/home/syam/ces745/openmp/Fortran/data-scope/openmp_lib_env-2.f90

```
…
write(*,*) "changes before"
call omp_set_dynamic(.TRUE.)
call omp_set_nested(.TRUE.)
!$omp parallel private(myid)
    !$omp master
      nthreads = omp_get_num_threads()
      dynamics = omp_get_dynamic()
      nnested = omp_get_nested()
      write(*,*) "nthreads, dynamics, nnested : ", nthreads, dynamics, nnested
    !$omp end master
    myid = omp_get_thread_num()
    write(*,*) "myid : ", myid
!$omp end parallel
write(*,*) "after“
…..
```

```
[~] ./openmp_lib_env-2
 start
 nthreads, dynamics, nnested :          1 F F
 before
 nthreads, dynamics, nnested :          8 F F
 myid :          0
 myid :          2
 myid :          4
 myid :          1
 myid :          5
 myid :          6
 myid :          7
 myid :          3
 after
 changes before
 nthreads, dynamics, nnested :          8 T T
 myid :          2
 myid :          0
 myid :          4
 myid :          1
 myid :          3
 myid :          6
 myid :          7
 myid :          5
 after
```

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*
- Can the program use a different number of threads in each parallel region?
  - OMP_DYNAMIC TRUE || FALSE
- Will nested parallel regions create new teams of threads, or will they be serialized?
  - OMP_NESTED TRUE || FALSE

Intel compiler

```
[~/ces745/openmp/Fortran/data-scope] export OMP_NUM_THREADS=4
[~/ces745/openmp/Fortran/data-scope] ./openmp_lib_env-f90
 start
 nthreads, dynamics, nnested :          1 F F
 before
 nthreads, dynamics, nnested :          4 F F
 myid :          0
 myid :          1
 myid :          2
 myid :          3
 after
```

Intel compiler

```
[~/ces745/openmp/Fortran/data-scope] export OMP_DYNAMIC=TRUE
[~/ces745/openmp/Fortran/data-scope] export OMP_NESTED=TRUE
[syam@orc131:~/ces745/openmp/Fortran/data-scope] ./openmp_lib_env-f90
 start
 nthreads, dynamics, nnested :          1 T T
 before
 nthreads, dynamics, nnested :          4 T T
 myid :          1
 myid :          2
 myid :          0
 myid :          3
 after
```

Example - pi

Example: Calculating π

- **Numerical integration**

$$\int_0^1 \frac{4}{1+x^2}\,dx = \pi$$

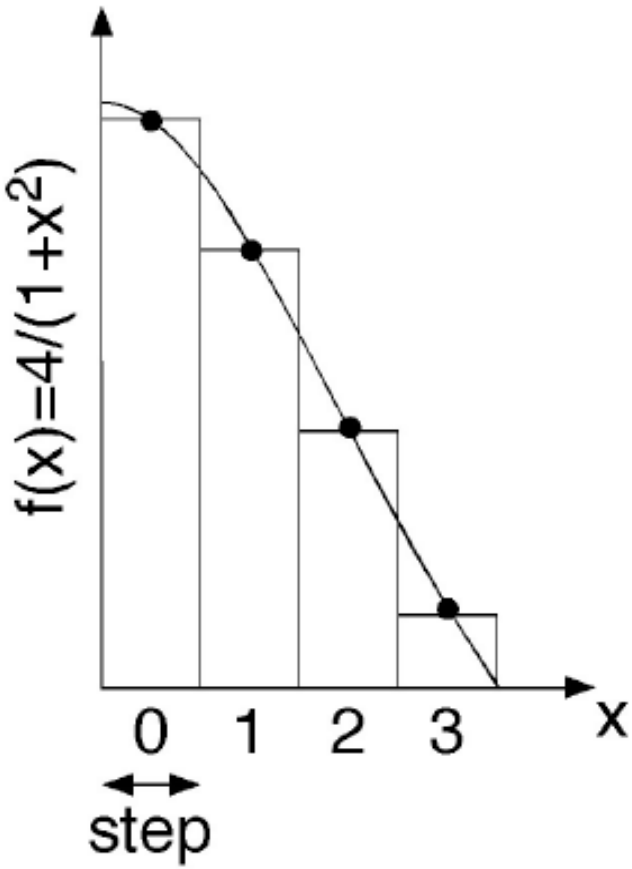
- **Discretization:**

$\Delta = 1/N$: `step = 1/NBIN`

$x_i = (i+0.5)\Delta \ (i = 0,\ldots,N-1)$

$$\sum_{i=0}^{N-1} \frac{4}{1+x_i^2}\Delta \cong \pi$$

```
#include <stdio.h>
#define NBIN 100000
void main() {
  int i; double step,x,sum=0.0,pi;
  step = 1.0/NBIN;
  for (i=0; i<NBIN; i++) {
    x = (i+0.5)*step;
    sum += 4.0/(1.0+x*x);}
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```



OpenMP Program: `omp_pi_critical.c`

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main() {
  double step,sum=0.0,pi;
  step = 1.0/NBIN;
#pragma omp parallel
  {
    int nthreads,tid,i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
      x = (i+0.5)*step;
#pragma omp critical
      sum += 4.0/(1.0+x*x);
    }
  }
  pi = sum*step;
  printf("PI = %f\n",pi);
}
```

Shared variables

Private (local) variables

This has to be atomic

~syam/ces745/openmp/C/pi/source/pi-critical.c

Avoid Critical Section: `omp_pi.c`

```
#include <stdio.h>
#include <omp.h>
#define NBIN 100000
#define MAX_THREADS 8
void main() {
  int nthreads,tid;
  double step,sum[MAX_THREADS]={0.0},pi=0.0;
  step = 1.0/NBIN;
#pragma omp parallel private(tid)
  {
    int i;
    double x;
    nthreads = omp_get_num_threads();
    tid = omp_get_thread_num();
    for (i=tid; i<NBIN; i+=nthreads) {
      x = (i+0.5)*step;
      sum[tid] += 4.0/(1.0+x*x);
    }
  }
  for(tid=0; tid<nthreads; tid++) pi += sum[tid]*step;
  printf("PI = %f\n",pi);
}
```

Array of partial sums for multi-threads

~syam/ces745/openmp/C/pi/source/pi-pregion.c

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];
            step = 1.0/(double) num_steps;
            omp_set_num_threads(NUM_THREADS);
#pragma omp parallel
{           double x;     int id;
            id = omp_get_thr ead_num();       sum[id] = 0;
#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;
}
```

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;
}
```

OpenMP adds 2 to 4 lines of code

```c
#include <omp.h>                /* OpenMP header file*/
#define NUM_STEPS 100000000

int main(int argc, char *argv[]) {
    int i;
    double x, pi;
    double sum = 0.0;
    double step = 1.0/(double) NUM_STEPS;
    int nthreads;
    /* do computation -- using all available threads */
    #pragma omp parallel
    {
        #pragma omp master
        {
            nthreads = omp_get_num_threads();
        }
        #pragma omp for private(x) reduction(+:sum) schedule(runtime)
        for (i=0; i < NUM_STEPS; ++i) {
            x = (i+0.5)*step;
            sum = sum + 4.0/(1.0+x*x);
        }
        #pragma omp master
        {
            pi = step * sum;
        }
    }
    /* print results */
    printf("parallel program results with %d threads:\n", nthreads);
    printf("pi = %g  (%17.15f)\n",pi, pi);
    return EXIT_SUCCESS;
}
```

pi-parallel_timing.c

OpenMP: Performance Issues

Key Factors that impact performance

- Parallel overheads
- Granularity
- Load balancing
- Locality
- Synchronization

Software/Programming issues

Highly tied with Hardware

Jacobi Solver – Serial code with 2 nested loops

```
k = 1;
 while (k <= maxit && error > tol) {
      error = 0.0;
      /* copy new solution into old */
   for (j=0; j<m; j++)
        for (i=0; i<n; i++)
             uold[i + m*j] = u[i + m*j];

      /* compute stencil, residual and update */
      for (j=1; j<m-1; j++)
        for (i=1; i<n-1; i++){
             resid =(
                          ax * (uold[i-1 + m*j] + uold[i+1 + m*j])
                          + ay * (uold[i + m*(j-1)] + uold[i + m*(j+1)])
                          + b * uold[i + m*j] - f[i + m*j]
                     ) / b;

             /* update solution */
             u[i + m*j] = uold[i + m*j] - omega * resid;

             /* accumulate residual error */
             error =error + resid*resid;
        }
      /* error check */
      k++;
   error = sqrt(error) /(n*m);

 } /* while */
```

```
cd ~syam/ces745/openmp/jacobi/c
icc -qopenmp -O2 driver.c realtime.c jacobi.c -o jacobi
```

Run on graham: /home/syam/ces745/openmp/jacobi/c

```
[~/ces745/openmp/jacobi/c] ./jacobi < input.large
Input n,m - grid dimension in x,y direction :
 Input alpha - Helmholts constant :
Input relax - Successive over-relaxation parameter:
 Input tol - error tolerance for iterrative solver:
Input mits - Maximum iterations for solver:
 -> 5000, 5000, 0.8, 1, 1e-07, 1000
Total Number of Iterations 2
Residual                    0.000000000038394
 elapsed time :     0.160518
 MFlops       :     -10372.4
Solution Error : 0.000106635
```

Jacobi Solver – Parallel code with 2 parallel regions

~syam/ces745/openmp/jacobi/c/jacobi_omp1.c

```
k = 1;
 while (k <= maxit && error > tol) {
      error = 0.0;
      /* copy new solution into old */
#pragma omp parallel for private(i)
   for (j=0; j<m; j++)
        for (i=0; i<n; i++)
             uold[i + m*j] = u[i + m*j];

      /* compute stencil, residual and update */
#pragma omp parallel for reduction(+:error) private(i,resid)
      for (j=1; j<m-1; j++)
        for (i=1; i<n-1; i++){
             resid = (  ax * (uold[i-1 + m*j] + uold[i+1 + m*j])
                        + ay * (uold[i + m*(j-1)] + uold[i + m*(j+1)])
                        + b * uold[i + m*j] - f[i + m*j]
                   ) / b;

             /* update solution */
             u[i + m*j] = uold[i + m*j] - omega * resid;

             /* accumulate residual error */
             error =error + resid*resid;
        }
      /* error check */
      k++;
     error = sqrt(error) /(n*m);
 } /* while */
```

Run on graham: /home/syam/ces745/openmp/jacobi/c

```
[~/ces745/openmp/jacobi/c] ./jacobi_omp1 < input.large
Input n,m - grid dimension in x,y direction :
 Input alpha - Helmholts constant :
Input relax - Successive over-relaxation parameter:
 Input tol - error tolerance for iterrative solver:
Input mits - Maximum iterations for solver:
 -> 5000, 5000, 0.8, 1, 1e-07, 1000
Total Number of Iterations 2
Residual                    0.000000000038394
 elapsed time :     0.063037
 MFlops       :     -23993.9
Solution Error : 0.000106635
```

Jacobi Solver – Parallel code with 2 parallel loops in 1 PR

~syam/ces745/openmp/jacobi/c/jacobi_omp2.c

```
k = 1;
while (k <= maxit && error > tol) {
    error = 0.0;
#pragma omp parallel private(resid, i)
{
#pragma omp for
  for (j=0; j<m; j++)
      for (i=0; i<n; i++)
          uold[i + m*j] = u[i + m*j];

    /* compute stencil, residual and update */
#pragma omp for reduction(+:error)
    for (j=1; j<m-1; j++)
      for (i=1; i<n-1; i++){
          resid = (  ax * (uold[i-1 + m*j] + uold[i+1 + m*j])
                     + ay * (uold[i + m*(j-1)] + uold[i + m*(j+1)])
                     + b * uold[i + m*j] - f[i + m*j]) / b;

          /* update solution */
          u[i + m*j] = uold[i + m*j] - omega * resid;
          /* accumulate residual error */
          error =error + resid*resid;
      }
  } /* end parallel */
    k++;
   error = sqrt(error) /(n*m);
} /* while */
```

Run on graham: /home/syam/ces745/openmp/jacobi/c

```
[~/ces745/openmp/jacobi/c] ./jacobi_omp2 < input.large
Input n,m - grid dimension in x,y direction :
 Input alpha - Helmholts constant :
Input relax - Successive over-relaxation parameter:
 Input tol - error tolerance for iterrative solver:
Input mits - Maximum iterations for solver:
 -> 5000, 5000, 0.8, 1, 1e-07, 1000
Total Number of Iteratuons 2
Residual                    0.000000000038394
 elapsed time :     0.042585
 MFlops       :     -39390.8
Solution Error : 0.000106635
```

Load Balance

Example: Sparse matrix

Data is not uniformly distributed, one thread will get more points than another.

Solution: Dynamic schedule

If load balancing is the most important issue to performance, perhaps we should use dynamic scheduling.

However, dynamic scheduling costs more than static:

1) more synchronization cost: each thread needs to go to the runtime library after each iteration and ask for another iteration to execute.

   Increasing the chunk size can reduce the synchronization overheads, but it will worsen the load balance.

2) data locality (distance in the cache, etc)

Load Balance: continue

Example: dense triangle matrix-scaling

```
for (i=0; i < n; i++){
    for (j=i; j<n; j++){
        a[i][j] = c* a[i][j]
    }
}
```

Each iteration has a different amount of work, but the amount of work varies regularly

Each successive iteration has a linearly decreasing amount of work

Solution: static schedule with a relatively small chunk size

Locality

The Memory Hierarchy

- Most parallel systems are built from CPUs with a memory hierarchy
  - Registers
  - Primary cache
  - Secondary cache
  - Local memory
  - Remote memory - access through the interconnection network
- As you move down this list, the time to retrieve data increases by about an order of magnitude for each step.
- Therefore:
  - Make efficient use of local memory (caches)
  - Minimize remote memory references

Performance Tuning - Cache Locality

- The basic rule for efficient use of local memory (caches):

  Use a memory stride of one

- This means array elements are accessed in the same order they are stored in memory.
- Fortran: "Column-major" order
  - Want the leftmost index in a multi-dimensional array varying most rapidly in a loop
- C: "Row-major" order
  - Want rightmost index in a multi-dimensional array varying most rapidly in a loop
- Interchange nested loops if necessary (and possible!) to achieve the preferred order.

Column major arrays vs. row major arrays

A two dimentional array like A[3][3]:

A11 A12 A13
A21 A22 A23
A31 A32 A33

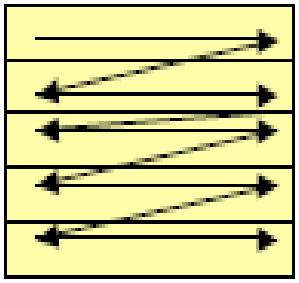
Main memory is just like a big 1D array with indices from 0x0 to 0Xffffff

This is **FORTRAN**'s column major order in memory:
A11 A21 A31 A12 A22 A32 A13 A23 A33

This is **C/C++**'s row major order in memory:
A11 A12 A13 A21 A22 A23 A31 A32 A33

Which of the following is faster in C?

```
for (i=0; i < 10000; i++)
  for (j=0; j < 10000; j++)
    sum += a[i][j];
```



```
for (j=0; j < 10000; j++)
  for (i=0; i < 10000; i++)
    sum += a[i][j];
```