Parallelism using OpenMP

Kevin Olson Ning Li

May 3, 2017

Roadmap

Roadmap for these webinars

1. Introduction to Xeon Phi
2. *Parallelism with OpenMP*
3. Vectorization with OpenMP
4. Hands-on exercise
5. Xeon Phi Tuning - Part 1
6. Xeon Phi Tuning - Part 2
7. Hands-on exercise

Introduction

- Here you will learn how to code for multi-core platforms
- There are many tools capable of achieving this (e.g. MPI, TBB, CoArray Fortran, Fortran 2008 (DO CONCURRENT), UPC also work on Xeon Phi processor).
- We only show OpenMP because it provides the easiest path to getting optimal performance from the Xeon Phi processor and because of time constraints.
- For the first time, Xeon Phi processor makes OpenMP competitive with other programming models (in our experience).
- And its portable (but not as portable as MPI).

Multi-core Parallelism

- For a long time neither C nor Fortran had any parallelism semantics (except for special purpose, architecture dependent versions)
- OpenMP extends the languages with the tools necessary for parallelism
- It also provides features to help codes perform across architectures

Learning Objectives

After this you will be able to

- Use OpenMP to write multi-core parallel code
- Use OpenMP features for performance portability
- Parallelize existing serial code with OpenMP

Sequence of topics

- This will focus on the most common kind of OpenMP parallelism
- It will also show the most common kinds of inter-thread communication (e.g the reduction)
- Then you will learn the building blocks these are actually made of
- We close with advanced topics which are important for performance

Xeon Phi processor is a Shared Memory Machine

- A shared memory "machine" is an environment where a collection of processors or cores can all access the same memory.
- All the processing cores of the Xeon Phi processor can read and write to the same memory space and so falls into this category.
- In general, the Xeon Phi processor is a NUMA (Non-Uniform Memory Access) machine and is Cache Coherent.
- All processors (cores) share the same memory, but each has its own cache.
- OpenMP is a prefered programming model for such machines.

What is OpenMP ?

- OpenMP uses threads to execute code in parallel.
- A thread is an execution entity, something that executes instructions independently.
- Each thread can have its own, private memory.
- An OpenMP thread is one that is managed by the OpenMP runtime system.
- A *thread safe* routine is one that functions correctly even when executed concurrently by multiple threads.

What is OpenMP ?

- OpenMP is a specification for shared memory parallelism.
- It is a mechanism for writing multithreaded code for shared memory machines.
- Extends C/C++ and Fortran through the use of Compiler Directives, Environment Variables and runtime Library Routines.

Other Sources of Information

- http://www.openmp.org - specification and links to forum.
- man pages
- Google (naturally)
- Books:
  - ”Parallel Programming in OpenMP” by Rohit Chandra et al.
  - “Using OpenMP” by Barbara Chapman et al.

OpenMP Feature Set

- Parallel Construct
- Work-Sharing Constructs (Loop, Section, Single, Workshare (Fortran only))
- Data-Sharing, No Wait, and Schedule Clauses
- Synchronization Constructs (Barrier, Critical, Atomic, Locks, Master)

PARALLEL Construct

Parallel Regions

- Parallel region is a block of code that will be executed in parallel on different threads
- Fortran example:

```
!$OMP PARALLEL [clause1 clause2 ...]
  ! block of code executed in parallel
!$OMP END PARALLEL
```

- Notes: No “GOTO” allowed, Must appear in same the same routine, STOP statement OK

Parallel Regions

- C/C++ example (Note: parallel region ends implicitly at end of structured block enclosed by braces)

```
#pragma omp parallel [clause1 clause2 ...]
{
  /* block of code executed in parallel */
}
```

- Notes: structured-block must have a single entry point at the top and bottom, no jumps into or out of block, call to exit() OK

OpenMP WORK SHARING CONSTRUCTS

Work Sharing Constructs

- Used to divide up the work in a parallel region
- No new threads launched by these constructs
- All threads in the current team must reach the work sharing construct
- There is an implied barrier on entry to these constructs, and it can be removed on exit from the construct.
- Loops, Sections, Workshare(Fortran 90 array operations), Single

OpenMP Parallel For/Do

- This directive instructs OpenMP to parallelize "for" loops with iterations distributed across the threads.
- Must be a DO or for loop (not DO WHILE or while) and must be iterative.
- Loops must complete, i.e. no branching out is allowed.
- It does a lot of work behind the scenes for performance
  - You will learn later how to change its behavior
- For now let's start with an example

OpenMP Parallel For: C Example

```

//Variables declared above
//directive become "shared"
#pragma omp parallel for
for(int i=0;i<n;i++){
  //Variables declared
  //inside code block
  //become "private"
}
```

OpenMP Parallel Do: Fortran Example

```fortran
!Variables declared above
!directive become "shared"
!$OMP PARALLEL DO
do i = 1, n
  !Variables can not be declared
  !inside code block
  !must use "private" clause in Fortran
end do
!$OMP END PARALLEL DO
```

Sections Construct

- Allows threads to execute different blocks of code in parallel
- Only 1 thread per "section"
- Can use `private`, `firstprivate`, `lastprivate`, `reduction`, and `nowait` clauses with "sections"

Sections Construct: C Example

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    {
      (void) funcA();
    }
    #pragma omp section
    {
      (void) funcB();
    }
  }
}
```

Sections Construct: Fortran Example

```fortran
!$OMP PARALLEL
  !$OMP SECTIONS
     !$OMP SECTION
     call subA()

     !$OMP SECTION
     call subB()

  !$OMP END SECTIONS
!$OMP END PARALLEL
```

Workshare Construct: Fortran Only

- Like parallel do construct
- Divides work up for Fortran array syntax
- Used with array assignements, `FORALL`, `WHERE`

Workshare Construct: Example

```
!$OMP PARALLEL SHARED(n, a, b, c)
!$OMP WORKSHARE
  A(1:n) = A(1:n) + 1
  B(1:n) = B(1:n) + 2
  C(1:n) = A(1:n) + C(1:n)
!$OMP END WORKSHARE
!$OMP END PARALLEL
```

Single Construct

- Allows only 1 thread to execute code block following it
- Other threads wait until single construct is executed
- First thread to "single" will execute code block

Single Construct: C Example

```c
#pragma omp parallel
{
  #pragma omp single
  {
    a = 10;
    printf("Single␣executed␣by␣%d\n",
           omp_get_thread_num());
  } /* Implicit barrier here */

  #pragma omp for
  for (i=0; i<n; i++) {
    b[i] = a;
  }
}
```

Single Construct: Fortran Example

```fortran
use omp_lib
!$OMP PARALLEL
  !$OMP SINGLE
  a = 10
  print *,'Single␣Construct␣excuted␣by␣',  &
       omp_get_thread_num()
  !$OMP END SINGLE

  !$OMP DO
  Do i = 1, n
     b(i) = a
  End Do
  !$OMP END DO

!$OMP END PARALLEL
```

OpenMP CLAUSES

Clauses

- OpenMP directives can take options in the form of clauses.
- In the case of the parallel directive they control the access to variables, decide how certain operations are performed or extend the functionality of the directive.
- While OpenMP programs act on shared data we will see that sometimes we need variables that have thread-specific values
- This includes the threads unique number or loop counters.
- Thus threads can have their own private copies of some variables
- We can control access to variables by means of the data sharing clauses to the parallel region.

Clauses for Parallel construct

- `private`
- `shared`
- `default`
- `if`
- `firstprivate, lastprivate`
- `num_threads`
- `reduction`
- `copyin`

Shared Clause

- Specifies which data is shared among threads
- Ensures that there is only one copy (or instance) of the data across all threads
- Each thread can read and modify this data (BE CAREFUL !!!)
- By default all variables are shared EXCEPT for iteration variables

Shared Clause: C Example

```
float x;
#pragma omp parallel for shared(x)
for(int i=0;i<n;i++){
  x=1.0;
  //Now if we modify "x"
  //that change is visible
  //to all threads

  int id=omp_get_thread_num();
  if(id==2)x=x+5.0;
#pragma omp barrier
  if(id==3)
    std::cout<<"x="<<x<<std::endl;
}
```

```
x=6.0
```

Shared Clause: Fortran Example

```fortran
program shared
  use omp_lib
  double precision :: x
!$OMP PARALLEL DO SHARED(x)
  do i = 0, n
     x = 1.0
     ! Now if we modify "x" that change is
     ! visible to all threads
     id = omp_get_thread_num()
     if (id == 2) x = x + 5.0
!$OMP BARRIER
     if (id == 3) print *,'␣x␣=␣',x
  end do
!$OMP END PARALLEL DO
end program shared
```

Private Clause

- Private clause forces each thread to have a copy of variables in its list
- Changes made to one variable that is private are visible ONLY to that thread
- By default iteration variables in loops are private
- By default, C/C++ variables declared inside a parallel region are private

Private Clause: C Example

```
float x;
#pragma omp parallel for private(x)
for(int i=0;i<n;i++){
  x=1.0;
  //Now if we modify "x"
  //That change
  //is localized
  //to that thread only
  int id=omp_get_thread_num();
  if(id==2)x=x+5.0;
#pragma omp barrier
  if(id==3)
    std::cout<<"x="<<x<<std::endl;
}
```

```
x=1.0
```

Private Clause: Fortran Example

```fortran
  program priv
    use omp_lib
    double precision :: x
!$OMP PARALLEL DO PRIVATE(x)
    do i = 1, n
       x = 1.0
       ! Now if we modify "x" that change is
       ! localized to that thread only
       id = omp_get_thread_num()
       if (id == 2) x = x + 5.0
!$OMP BARRIER
       if (id == 3) print *,'␣x␣=␣',x
    end do
!$OMP END PARALLEL DO
  end program priv
```

Firstprivate and Lastprivate Clauses

- Simliar to Private clause EXCEPT:
- Firstprivate: Each variable in list takes on a pre-initialized value taken from the serial region on the 'master' thread.
- Lastprivate:
  - for a DO or for loop, the value becomes that computed in the sequentially last iteration
  - for sections, the value becomes that in the lexically last section
  - values not assigned a value by the last DO/for iteration or lexically last section become undefined

Default Clause

- Used to give variables a default data-sharing attribute
- Syntax: `default(none | private | shared)`
- Recommended to use `default(none)` and then explicitly specify shared and private variables

Nowait Clause

Causes construct it is associated with to NOT have an implicit barrier, threads continue executing without stopping and waiting

```
#pragma omp for nowait
for (i = 0; i<n; i++)
  {
     /* some code */
  }
```

```
!$OMP DO
! some fortran code
!$OMP END DO NOWAIT
```

Use of nowait clause is recommended.

Schedule Clause

- Can be used with for/do loop construct only
- syntax: `schedule(kind,` *[chunk_size]* `)`
- `kind` can take on the values:
  - `static`: Loop chunks are assigned statically in round-robin manner in the order of the thread number
  - `dynamic`: Loop chunks are assigned to threads as the threads become available and request them
  - `guided`: Loop chunks are assigned to threads as the threads request them, but the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads.
  - `runtime`: scheduling is done at runtime and is influenced by `OMP_SCHEDULE` environment variable

If clause

- Supported on "parallel" construct only
- Syntax: `if(logical-expression)`
- if `logical-expression` evaluates to true, then team of threads executes the following code block
- if evaluates to false, parallel region is executed by 1 thread only

num_threads clause

- Supported on "parallel" construct only
- Syntax: `num_threads (n)`
- `n` is an integer that specifies the number of threads that will execute the parallel region

Reduction Clause

- Works on shared data
- Used for evaluating recurrence operations on mathematically associative and commutative operators
- A global sum the most common example where this is used
- C Syntax: `reduction(`*`operator:list`*`)`
- Fortran Syntax: `reduction(`*`operator`*` | `*`intrinsic_procedure:list`*`)`

Reduction operators for C

Operator	Initial Value
+	0
*	1
-	0
&	∼0
|	0
∧	0
&&	1
||	0

Reduction operators for Fortran

Operator	Initial Value
+	0
*	1
-	0
`.and.`	`.true.`
`.or.`	`.false.`
`.eqv.`	`.true.`
`.neq.`	`.false.`
`.neqv.`	`.false.`

Reduction instrinsics for Fortran

Instrinsic	Initial Value
`max`	Smallest neg. value
`min`	Largest pos. value
`iand`	All bits one
`ior`	0
`ieor`	0

How do we get this to work?

```
int n=500;
vector<float> x(n,1.0);

float tmp=0.0;
#pragma omp parallel for shared(tmp)
for(int i=0;i<n;i++){
  tmp+=x[i];
}
std::cout<<"sum(x)="<<tmp<<std::endl;
```

```
sum=439
```

OpenMP Parallel Reduction:Example

```
int n=500;
vector<float> x(n,1.0);

float tmp=0.0;
#pragma omp parallel for reduction(+:tmp)
for(int i=0;i<n;i++){
  tmp+=x[i];
}
std::cout<<"sum(x)="<<tmp<<std::endl;
```

```
sum=500
```

BREAK

SYNCHRONIZATION CONSTRUCTS

Synchronization Constructs

- barrier
- ordered
- critical
- atomic
- locks (not dicussed)
- master

Barrier Construct

- Inserted inside of parallel regions
- All threads wait at barrier until all other threads reach the barrier, then execution continues
- All threads MUST reach the barrier, or code locks up
- C syntax: `#pragma omp barrier`
- Fortran syntax: `!$OMP_BARRIER`

Critical Regions

- Critical regions are blocks which only permit one executing thread at a time
- Critical regions effectively serialize code
- So they are very inefficient
- For best performance: minimize their use

Critical Regions: Example

```
int n=500;
vector<float> x(n,1.0);

float tmp=0.0;
#pragma omp parallel for shared(tmp)
for(int i=0;i<n;i++){
#pragma omp critical
  tmp+=x[i];
}
std::cout<<"sum(x)="<<tmp<<std::endl;
```

```
    sum=500
```

Better to use "reduction" clause

Critical Regions: Example

```c


FILE* fp=fopen("filename.txt","w");
#pragma omp parallel for
for(int i=0;i<n;i++){
  float x=long_computation(i);
#pragma omp critical
  fprintf(fp,"%f\n",x);
}
```

Master Construct

- Allows only the first thread allocated at program initiation (the "master") to execute block of code
- C Syntax:

```
#pragma omp master
{
  /* structured block */
}
```

- Fortran Syntax:

```
!$OMP MASTER
  ! Structured block
!$OMP END MASTER
```

Ordered Construct

- Forces a block of code in a parallel loop to execute in sequential order
- Code is not serialized, but loop behaves as if it was executed serially

```c
#pragma omp ordered
{
  /* block of code to execute sequentially */
}
```

```fortran
!$OMP ORDERED
  ! structured block of code to execute sequentially
!$OMP END ORDERED
```

Atomic Construct

- Allows all threads to update shared data without interference, works almost like "critical" construct
- Can be an efficient alternative to "critical" IF hardware supports atomic operations (Xeon Phi processor does!)
- If hardware supports atomic ops., atomic construct uses them (i.e. reads from memory, modify value, and write back all in one action)
- C syntax:

```c
#pragma omp atomic
/* single statement (can be a function call) */
```

- Fortran Syntax:

```fortran
!$OMP ATOMIC
  ! single statement
```

Atomic Example

```c
int ic, i, n;
ic = 0;
#pragma omp parallel default(none) \
        shared(n,ic) private(i)
for (i=0; i<n; i++)
  {
    #pragma omp atomic
    ic += bigfunc();
  }
printf("counter = %d\n", ic);
```

Update of `ic` occurs atomically, `bigfunc()` can execute at the same time on all threads

OpenMP Environment Variables

- Environment variables are read before any OpenMP construct or routine. Changing the value of an environment variable will have no effect afterwards.
- The behaviour of the program can be investigated or changed after this point with run-time library routines.
-

Environment Variables

`OMP_NUM_THREADS`

- This is used to explicitly set number of threads for parallel regions.
- The behavior of the program is implementation defined if the values lead to a number of threads it can not support.

`OMP_SCHEDULE`

- This variable is used for DO and for directives specifying the `schedule(runtime)` clause.
- Specify the type of scheduling and chunk size if appropriate. For example: `export OMP_SCHEDULE=static,1`

Runtime Library Routines

- `use omp_lib` is needed in modules that use these functions
- `subroutine omp_set_num_threads(num)`
  `integer num`
  Sets the number of threads for subsequent parallel regions. Must be a positive integer.
- `integer function omp_get_num_threads()`
  Gets the current number of threads. The routine will return 1 outside of all parallel regions.
- `integer function omp_get_max_threads()`
  Returns the maximum number of threads allowed for a new team of threads. It is an upper bound on the number of threads to use in a parallel region without a num_threads clause. You could use the value to allocate sufficient storage.

Runtime Library Routines

- `integer function omp_get_thread_num()`
  Get the thread's unique number in the current team.
- `integer function omp_get_num_procs()`
  Gets the number of processors available to an OpenMP program.
- `double precision omp_get_wtime()` returns wall time.
- `subroutine omp_set_schedule(sched, chunk)`
  `integer (kind=omp_sched_kind) sched`
  `integer chunk`
  Override the value set with the `OMP_SCHEDULE` environment variable for runtime scheduling.

Runtime Library Routines

- `subroutine omp_get_schedule(sched, chunk)`
  `integer (kind=omp_sched_kind) sched`
  `integer chunk`
  Get the current value for runtime scheduling.
- Values for kind can be one of the following constants or the equivalent numerical values:
  - ☐ `omp_sched_static = 1`
  - ☐ `omp_sched_dynamic = 2`
  - ☐ `omp_sched_guided = 3`
  - ☐ `omp_sched_auto = 4`
- The value of modifier gives the chunk size if applicable.
- The default is used for modifier <1.

Runtime Library Routines: C/C++

- `#include<omp.h>`
- `void omp_set_num_threads(int)`
- `int omp_get_num_threads(void)`
- `int omp_get_max_threads(void)`
- `int omp_get_thread_num(void)`
- `int omp_get_num_procs(void)`
- `double omp_get_wtime(void)`
- `void omp_set_schedule(omp_sched_t sched, int chunk)`
- `void omp_get_schedule(omp_sched_t *sched, int *chunk)`

ADVANCED OpenMP FOR PERFORMANCE

General OMP Best Practices

First, use what we've covered in the best way possible:

- Maximize Parallel regions, avoid using inside inner loops
- Avoid Barriers, Ordered, Critical and Locks
- Load Balance
- Use “nowait” clause
- Parallelize outer loops, vectorize inner loops (more later)

Advanced OpenMP: Achieving Performance

- One great aspect of OpenMP is its tunability
- Well written OpenMP code should perform well across architectures
  - Modulo minor parameter changes
- These parameters vary how threads execute a parallel region

More about Scheduling

- Scheduling plus chunking dictates how iterates of loop are divided into threads
- OpenMP gives considerable flexibility in this
- Best illustrated by example

Scheduling: Static

```
int nthreads=4;
int n=16;
vector<int> ids(n,0);
#pragma omp parallel for num_threads(4) \
            schedule(static)
for(int i=0;i<n;i++){
  int id=omp_get_thread_num();
  ids[i]=id;
}

for(int i=0;i<n;i++){
  std::cout<<"(id,i)="<<"("<<ids[i]<<","<<i<<")\n";
}
```

Scheduling: Static

```
(id,i)=(0,0)
(id,i)=(0,1)
(id,i)=(0,2)
(id,i)=(0,3)
(id,i)=(1,4)
(id,i)=(1,5)
(id,i)=(1,6)
(id,i)=(1,7)
(id,i)=(2,8)
(id,i)=(2,9)
(id,i)=(2,10)
(id,i)=(2,11)
(id,i)=(3,12)
(id,i)=(3,13)
(id,i)=(3,14)
(id,i)=(3,15)
```

Scheduling: Static

```
int nthreads=4;
int n=16;
vector<int> ids(n,0);
#pragma omp parallel for num_threads(4) \
            schedule(static,1)
for(int i=0;i<n;i++){
  int id=omp_get_thread_num();
  ids[i]=id;
}

for(int i=0;i<n;i++){
  std::cout<<"(id,i)="<<"("<<ids[i]<<","<<i<<")\n";
}
```

Scheduling: Static, chunk size = 1

```
    (id,i)=(0,0)
    (id,i)=(1,1)
    (id,i)=(2,2)
    (id,i)=(3,3)
    (id,i)=(0,4)
    (id,i)=(1,5)
    (id,i)=(2,6)
    (id,i)=(3,7)
    (id,i)=(0,8)
    (id,i)=(1,9)
    (id,i)=(2,10)
    (id,i)=(3,11)
    (id,i)=(0,12)
    (id,i)=(1,13)
    (id,i)=(2,14)
    (id,i)=(3,15)
```

Scheduling: Static

```
int nthreads=4;
int n=16;
vector<int> ids(n,0);
#pragma omp parallel for num_threads(4) \
            schedule(static,2)
for(int i=0;i<n;i++){
  int id=omp_get_thread_num();
  ids[i]=id;
}

for(int i=0;i<n;i++){
  std::cout<<"(id,i)="<<"("<<ids[i]<<","<<i<<")\n";
}
```

Scheduling: Static, chunk size = 2

```
(id,i)=(0,0)
(id,i)=(0,1)
(id,i)=(1,2)
(id,i)=(1,3)
(id,i)=(2,4)
(id,i)=(2,5)
(id,i)=(3,6)
(id,i)=(3,7)
(id,i)=(0,8)
(id,i)=(0,9)
(id,i)=(1,10)
(id,i)=(1,11)
(id,i)=(2,12)
(id,i)=(2,13)
(id,i)=(3,14)
(id,i)=(3,15)
```

Scheduling: Dynamic

```
int nthreads=4;
int n=16;
vector<int> ids(n,0);
#pragma omp parallel for num_threads(4) \
            schedule(dynamic)
for(int i=0;i<n;i++){
  int id=omp_get_thread_num();
  ids[i]=id;
}

for(int i=0;i<n;i++){
  std::cout<<"(id,i)="<<"("<<ids[i]<<","<<i<<")\n";
}
```

Scheduling: Guided

```
int nthreads=4;
int n=16;
vector<int> ids(n,0);
#pragma omp parallel for num_threads(4) \
            schedule(guided)
for(int i=0;i<n;i++){
  int id=omp_get_thread_num();
  ids[i]=id;
}

for(int i=0;i<n;i++){
  std::cout<<"(id,i)="<<"("<<ids[i]<<","<<i<<")\n";
}
```

Scheduling: Auto

```
int nthreads=4;
int n=16;
vector<int> ids(n,0);
#pragma omp parallel for num_threads(4) \
        schedule(auto)
for(int i=0;i<n;i++){
  int id=omp_get_thread_num();
  ids[i]=id;
}

for(int i=0;i<n;i++){
  std::cout<<"(id,i)="<<"("<<ids[i]<<","<<i<<")\n";
}
```

Scheduling

- Your choices are:
  - Complete control (static)
  - Less control (Dynamic,Guided,Auto)
- Don't conclude anything without first trying
- You may be surprised which is best

Loop Collapsing

- Another possibility is that loop contains too few iterates
- With Xeon Phi processor we can have up to 256 hardware threads
- Loops can routinely have fewer than 256 iterates

Loop Collapsing

Note the outer loop in

```
int nthreads=256;
#pragma omp parallel for num_threads(256)
for(int i=0;i<2048;i+=512){
  for(int j=0;j<64;j++){
    long_computation(i,j);
  }
}
```

has only 4 iterates, but we ask for 256 threads

Loop Collapsing

The modified directive in

```c
int nthreads=256;
#pragma omp parallel for num_threads(256) \
            collapse(2)
for(int i=0;i<2048;i+=512){
  for(int j=0;j<64;j++){
    long_computation(i,j);
  }
}
```

fixes this by parallelizing over both outer and inner loop.

Affinity

- Up until now, parameters dealt with threads
- We have said little about the actual cores they execute on
- But OpenMP gives some degree of control of where threads execute on the hardware
- *Thread Affinity* helps control what resources a thread uses
- Usually it is used to prevent two threads from executing on one core
- But also useful to accomplish the *opposite*
- Setting the Intel environment variable `KMP_AFFINITY=[SCATTER | COMPACT]` can have a BIG impact (worth experimenting).

Affinity

- In OpenMP "places" are grouping of threads, cores, or sockets.
- Affinity can also be influenced by setting the environment variable `OMP_PLACES` (see OpenMP documentation for a discussion)
- OpenMP gives proc_bind clause for setting affinity, used with "parallel" construct
- Syntax: `proc_bind (master | close | spread)`
- `master`: all threads go to "place" of the master thread
- `close`: assign thread to "place" closest to place of parent thread
- `spread`: spread threads across "places"

Affinity: Example

```
#pragma omp parallel for proc_bind(spread)
for(int i=0;i<n;i++){
  //Threads will "spread"
  //which means they will
  //execute on unused
  //core before scheduling
  //on used core
}
```

Affinity: Example

```
#pragma omp parallel for proc_bind(close)
for(int i=0;i<n;i++){
  //Threads will schedule
  //"close" which means
  //they will seek to utilize
  //all availlable hardware threads
  //on a core before scheduling
  //on unused core
}
```

Putting It Together: Example

- Here you saw how to use OpenMP for parallelism
- One last example to show a common skeleton

```
#pragma omp parallel for \
  num_threads(nthreads) \
  collapse(CLPS) \
  proc_bind(BIND) \
  schedule(SCHTYPE,CHUNK)
for(int i=0;i<ni;i++){
  for(int j=0;j<nj;j++){
    //more nested loops..
  }
}
```

Practical Exercise

PARALLELIZING A CODE USING OpenMP