CIS 631
Parallel Processing

Lecture 9: Shared Memory
Parallel Programming and OpenMP

Allen D. Malony
malony@cs.uoregon.edu

Department of Computer and Information Science
University of Oregon
Acknowledgements

PORTIONS OF THE LECTURES SLIDES WERE ADOPTED FROM:

○ John Mellor-Crummey, COMP 422, “Parallel Programming,” Rice University, 2002
○ Vijay Pai, COMP 422, “Parallel Programming,” Rice University, 2002
○ A. Grama, A. Gupta, G. Karypis, and V. Kumar, “Introduction to Parallel Computing,” 2003
○ L. Kale, CS 320, “Parallel Computing,” University of Illinois, Urbana-Champaign, 2003
Outline

- OpenMP
References

- [http://www.openmp.org](http://www.openmp.org)
  - History of OpenMP
  - Current status and specifications
  - Tutorials
  - Everything you need to know!!!
OpenMP – What Is It?

- An API for Writing Multithreaded Applications
- A set of compiler directives and library routines for parallel application programmers
- Makes it easy to create shared memory parallel programs
  - Using Fortran, C, and C++
- Standardizes last 20+ years of SMP practice
- Supported by many hardware and software vendors
  - Intel, Microsoft, Cray, PGI, …
OpenMP – Programming Model

- Fork-Join Parallelism
  - Master thread spawns slave threads as needed
  - Parallelism is added incrementally
    - sequential program evolves into a parallel program
OpenMP – Thread Interaction

- This is shared memory parallel programming
  - Threads communicate by sharing variables
  - Unintended sharing of data can lead to race conditions

- Race condition
  - Program’s outcome depends on thread ordering
  - Typically not desired
  - Non-deterministic execution

- To control race conditions
  - Use synchronization to protect data conflicts
  - Synchronization is expensive
  - Change data storage to minimize need for synchronization
OpenMP – General Rules

- Most OpenMP constructs are *compiler directives*
- Directives inform the compiler
  - Provide compiler with knowledge
  - Usage assumptions
- Directives are ignored by non-OpenMP compilers!
  - Essentially act as comment for backward compatibility
- For C and C++, the syntax is:
  ```
  #pragma omp construct [clause [clause]...]
  ```
- For Fortran, the syntax is:
  ```
  !$OMP construct [clause [clause]...]
  !$OMP construct [clause [clause]...]
  !$OMP construct [clause [clause]...]
  ```
Structured Blocks

- Most OpenMP constructs apply to structured blocks
- Structured block
  - A block of code with one point of entry at the top and one point of exit at the bottom
  - Only other branches allowed out of the block are STOP statements in Fortran and exit() in C/C++
- Loops are a common example of structured blocks
PARALLEL Directive

- Specifies the following should be executed in parallel:
  - A program section (structured block)
  - If applied to a loop, what happens
    - iterations are executed in parallel
    - do loop (Fortran) or for loop (C/C++)

- PARALLEL DO is called a “worksharing” directive
  - Causes work to be shared across threads
  - More on this later
PARALLEL DO: Syntax

- Fortran

    !$omp parallel do [clause [,] [clause ...]]
    do index = first, last [, stride]
        body of the loop
    enddo
    !$omp end parallel do

- C/C++

    #pragma omp parallel for [clause [clause ...]]
    for (index = first; text_expr;
         increment_expr) {
        body of the loop
    }

Example: PARALLEL DO

☐ Single precision $a*x + y$ (saxpy)

```fortran
subroutine saxpyy (z, a, x, y, n)
    integer i, n
    real z(n), a, x(n), y(n)
    !$omp parallel do
    do i = 1, n
        z(i) = a * x(i) + y(i)
    enddo
    return
end
```
**Execution Model of PARALLEL DO**

- Master thread executes serial portion of code
- Master thread enters `saxpy` routine
- Master thread encounters *parallel do* directive
- Creates slave threads (How many?)
- Master and slave threads divide iterations of parallel `do` loop and execute them concurrently
- Implicit synchronization: wait for all threads to finish their allocation of iterations
- Master thread resumes execution after the `do` loop
- Slave threads disappear

☐ Abstract execution model
Loop-level Parallelization Paradigm

- Execute each loop in parallel
  - Where possible
- Easy to parallelize code
- Similar to automatic parallelization
- Incremental parallelization
  - One loop at a time
  - Does not break code (Really?)
- Fine-grain overhead
  - Frequent synchronization
- Performance determined by sequential part (Why?)

```c
C$OMP PARALLEL DO
do i=1,n
    ..........
enddo
alpha = xnorm/sum
C$OMP PARALLEL DO
do i=1,n
    ..........
enddo
C$OMP PARALLEL DO
do i=1,n
    ..........
enddo
```
Example: PARALLEL DO – Bad saxpy

- Single precision a*x + y (saxpy)

```fortran
subroutine saxpy (z, a, x, y, n)
integer i, n
real z(n), a, x(n), y(n)
!$omp parallel do
  do i = 1, n
    y(i) = a * x(i+1) + y(i+1)
  enddo
return
end
```

What happens here?
How Many Threads?

- Use environment variable
  - `setenv OMP_NUM_THREADS 8` (Unix machines)
- Use `omp_set_num_threads()` function

```fortran
subroutine saxpy (z, a, x, y, n)
  integer i, n
  real z(n), a, x(n), y(n)
  call omp_set_num_threads(4)
  !$omp parallel do
  do i = 1, n
    z(i) = a * x(i) + y(i)
  enddo
  return
end
```
Assigning Iterations to Threads

- A parallel loop in OpenMP is a worksharing directive.
- The manner in which iterations of a parallel loop are assigned to threads is called the loop’s schedule.
- Default schedule assigns iterations to threads as evenly as possible (good enough for saxpy).
- Alternative user-specified schedules possible.
- More on scheduling later.
Communication Between Threads

- Six clause types allow the programmer to specify how data (i.e., each variable) is shared between threads executing a parallel do:
  - private(list of variable/array names)
  - shared(list)
  - default(private | shared | none)
  - reduction(intrinsic operator : list)
  - firstprivate(list)
  - lastprivate(list)

- Data scope clauses
Default Data Sharing Rules

- Most variables are shared by default
  - This means there is only one copy shared by all threads
  - Global variables are *shared* among threads
    - Fortran: COMMON blocks, SAVE variables, MODULE variables
    - C/C++: File scope variables, static
- Stack variables in sub-programs called from parallel regions are *private*
  - Threads have their own private stack
- Automatic variables within a statement block are *private*
Private Clause

- Each thread has copy of all variables declared private
- Private variables (implicit or explicit) are uninitialized when a thread starts
  - Thread’s responsibility
- The value of a private variable is unavailable to the master thread after a parallel loop terminates
- Is there a way to pass private data back?
Firstprivate and Lastprivate

- **firstprivate** (list)
  - Initializes each thread’s copy of a private variable to the value of the master thread’s copy, for all variables in list.

- **lastprivate** (list)
  - Writes back to the master’s copy the value contained in the private copy belonging to the thread that executed the sequentially *last* iteration of the loop, for all variables in list.
Communication Between Threads

- Unless one of the data scope clauses is present, most data/variables are shared by default

```fortran
subroutine saxpy (z, a, x, y, n)
  integer i, n
  real z(n), a, x(n), y(n)
  !$omp parallel do
  do i = 1, n
    z(i) = a * x(i) + y(i)
  enddo
  return
end
```

- Is there a problem here?

- OpenMP will implicitly take care of the index variable
Example: Data Scope Clauses

double x, y;
int i, j, m, n, maxiter;
int depth[300][200];
extern int mandel_val();
n = 300;
m = 200;
maxiter = 200;
#pragma omp parallel for private(j, x, y)
for (i = 1; i <= m; i++)
    for (j = 1; j <= n; j++) {
        x = i/ (double) m;
        y = j/ (double) n;
        depth[j][i] =
            mandel_val(x, y, maxiter);
    }
Example: Private Clause

What is wrong with this example?

```fortran
program wrong
  IS = 0
  C$OMP PARALLEL DO PRIVATE(IS)
  DO J=1,100
    ...
    ... = IS
    ...
  100 CONTINUE
  print *, IS
```
Corrected Example

```
program wrong right
IS = 0
C$OMP PARALLEL DO FIRSTPRIVATE(IS)
C$OMP+ LASTPRIVATE(IS)
  DO J=1,100
    ... = IS
    ... = IS
    ...
  100 CONTINUE
print *, IS
```
Example: firstprivate, lastprivate

```fortran
common /mycom/ x, c, y, z
real x(n, n), c(n, n), y(n), z(n)
x(1, 1) = ...
x(2, 1) = ...
!
omp parallel do firstprivate(x) lastprivate(i, x)
do i = 1, n-1
   x(1, 2) = c(i, 1) * x(1, 1)
x(2, 2) = c(i, 2) * x(2, 1) ** 2
   y(i) = x(2, 2) + x(1, 2)
   z(i) = x(2, 2) - x(1, 2)
endo
doi+1) = x(1, 2) + x (2, 2)
```

What if you did not have lastprivate?

□
Example: reduction

subroutine sum (values, n, s)
  integer n, s
  real values(n)
  s = 0
  !$omp parallel do reduction(+ : s)
  do i = 1, n
    s = s + values(i)
  enddo
  return
end

☐ If s were shared, you’d need to protect it with locks

☐ If it were private, how do you get the global sum?

☐ Reduction clause simplifies your code
More on Reductions

- Operator must be commutative and associative
- Caution with floating point numbers and roundoff errors
  - $A + (B+C)$ may not be the same as $(A+B) +C$
Default clause

- Default storage attribute is `DEFAULT (SHARED)`
  - No need to specify
- `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
- C/C++ do not support `DEFAULT (PRIVATE)`
Example: DEFAULT Clause

\[ \text{itotal} = 1000 \]
\[
\text{C$OMP$ PARALLEL DO PRIVATE(np, each)}
\]
\[
\text{do } i = 1, 100
\]
\[
\text{np} = \text{omp\_get\_num\_threads()}
\]
\[
\text{each} = \text{itotal}/\text{np}
\]
\[
\ldots
\]
\[
\text{enddo}
\]

\[ \text{itotal} = 1000 \]
\[
\text{C$OMP$ PARALLEL DO DEFAULT(PRIVATE) SHARED(itotal)}
\]
\[
\text{do } i = 1, 100
\]
\[
\text{np} = \text{omp\_get\_num\_threads()}
\]
\[
\text{each} = \text{itotal}/\text{np}
\]
\[
\ldots
\]
\[
\text{enddo}
Assigning Iterations to Threads

- Motivation
  - Balance the work per thread
  - When the work per iteration is inherently unbalanced

```c
!$omp parallel do private(xkind)
  do i = 1, n
    xkind = f(i)
    if (xkind .lt. 10 then)
      call smallwork(x[i])
    else
      call bigwork(x[i])
    endif
  enddo
```

What happens if iterations are statically assigned to threads?
Schedule Clause

- `schedule (type[, chunk])`

- `type = static, dynamic, guided, runtime`  
- `chunk = scalar integer value`  
- `static`
  - Iterations are divided as evenly as possible among all threads  
    - *Simple static*  
- `static, chunk`
  - Iterations are divided into chunks of size `chunk`  
  - Chunks are then assigned in round robin fashion to threads  
    - *Interleaved*
Schedule Clause

- **dynamic, chunk**
  - Iterations are divided into chunks of size chunk (1 if unspecified) and are assigned to threads dynamically after an initial round robin assignment
  - *Simple dynamic* (chunk size of 1)

- **guided, chunk**
  - Chunk size decreases exponentially from an implementation dependent value (usually N/P) to chunk (1 if unspecified)
  - Chunks are assigned dynamically
  - *Guided self scheduling*
Schedule Clause

- **runtime**
  - The schedule type is chosen at runtime based on the environmental variable `OMP_SCHEDULE`
    ```bash
    setenv OMP_SCHEDULE "dynamic, 3"
    ```
- The choice of schedule is a tradeoff between load balancing and schedule overhead
  - Simple static has the minimum overhead, but has poor load balancing capability
  - Guided is expensive, but balances load better
Schedule Clause

- The best schedule varies based on the structure of each loop and can also vary based on the input data set.
- There was a lot of research work on loop scheduling in parallelizing compilers that motivated this.
  

Caution

The schedule clause is a tool for performance enhancement, not for ensuring program correctness.
PARALLEL DO: The Small Print

- The programmer has to make sure that the iterations can in fact be executed in parallel
  - No automatic verification by the compiler

```fortran
subroutine noparallel (z, a, x, y, n)
  integer i, n
  real z(n), a, x(n), y(n)
  !$omp parallel do
  do i = 2, n
    z(i) = a * x(i) + y(i) + z(i-1)
  enddo
  return
end
```
PARALLEL DO: Restrictions

- Number of times that the loop body is executed (trip-count) must be available at runtime before the loop is executed
  - Fortran:
    ```plaintext
    Do index = lowerbound, upperbound [, stride]
    trip count computable from bounds and stride
    ```
  - C/C++:
    ```plaintext
    for (index=start; index op end; incr_expr)
    index must be an integer variable
    op must be <, <=, > or >=
    expressions start and end must not change during execution
    ```
PARALLEL DO: Restrictions (continued)

- `increment_expr` must change the value of `index` by the same amount after each iteration.

- `increment_expr` must be of the form `index++`, `++index`, `index--`, `--index`, `index += incr`, `index -= incr`, `index = index + incr`, `index = incr + index`, or `index = index - incr`, where `incr` is an expression that does not change during the loop.

- Loop body must be able to complete all iterations:
  - Fortran: no exit or goto that branches outside the loop
  - C/C++: no break or goto that branches outside the loop
  - C++: no exception caught by a try block outside the loop
PARALLEL Directive

- Fortran
  
  !$omp parallel [clause [,] [clause ...]]
  
  structured block

  !$omp end parallel

- C/C++

  #pragma omp parallel [clause [clause ...]]
  
  structured block
Parallel Directive: Details

- When a parallel directive is encountered, threads are spawned which execute the code of the enclosed structured block (i.e., the parallel region)
- The number of threads can be specified just like for the parallel do directive
- The parallel region is replicated and each thread executes a copy of the replicated region
Example: Parallel Region

double A[1000];
omp_set_num_threads(4);
#pragma omp parallel
{
    int ID = omp_thread_num();
    pooh(ID, A);
}
printf(“all done
”);

double A[1000];
omp_set_num_threads(4)
|                     |
|                     |
| double A[1000];    |
|                     |
| omp_set_num_threads(4) |

ID = omp_thread_num()  ...  ID = omp_thread_num()
|                     |
| pooh(0,A)  pooh(1,A)  pooh(2,A)  pooh(3,A) |

printf(“all done
”);
Parallel versus Parallel Do

- Arbitrary structured blocks versus loops
- Coarse grained versus fine grained
- Replication versus work division

```c
!$omp parallel do
do I = 1,10
    print *, 'Hello world', I
enddo

!$omp parallel
do I = 1,10
    print *, 'Hello world', I
enddo
!$omp end parallel
```

Output: 10 Hello world messages

Output: 10*T Hello world messages where T = number of threads
Parallel Directive: Clauses

- private (list)
- shared (list)
- default (private | shared | none)
- reduction(intrinsic operator : list)
- if(logical_expression)
- copyin(list)
Parallel: Back to Motivation

```c
omp_set_num_threads(2);
#pragma omp parallel private(i, j, x, y, my_width,
    my_thread, i_start, i_end)
{
    my_width = m/2;
    my_thread = omp_get_thread_num();
    i_start = 1 + my_thread * my_width;
    i_end = i_start 1 + my_width - 1;
    for (i = i_start; i <= i_end; i++)
        for (j = 1; j <= n; j++) {
            x = i/ (double) m;
            y = j/ (double) n;
            depth[j][i] = mandel_val(x, y, maxiter);
        }
    for (i = i_start; i <= i_end; i++)
        for (j = 1; j <= n; j++)
            dith_depth[j][i] = 0.5*depth[j][i]
            + 0.25*(depth[j-1][i] + depth[j+1][i])
}
```
Work Sharing in Parallel Regions

- Manual division of work (previous example)
- OMP Work sharing constructs
  - Simplify the programmers job in dividing work among the threads that execute a parallel region
    - do directive
      - have different threads perform different iterations of a loop
    - sections directive
      - identify sections of work to be assigned to different threads
    - single directive
      - specify that a section of code is to be executed by one thread only (remember default is replicated)
DO Directive

- **Fortran**

  ```fortran
  !$omp parallel [clause [,] [clause ...]]
  ...
  !$omp do [clause [,] [clause ...]]
  do loop
  !$omp enddo [nowait]
  ...
  !$omp end parallel
  ```

- **C/C++**

  ```c++
  #pragma omp parallel [clause [clause ...]]
  {
      ...
      #pragma omp for [clause [clause] ... ]
      for-loop
  }
  ```
**DO Directive: Details**

- The DO directive does not spawn new threads - it just assigns work to the threads already spawned by the PARALLEL directive.
- The work↔thread assignment is identical to that in the PARALLEL DO directive.

```plaintext
!$omp parallel do
  do I = 1,10
    print *, 'Hello world', I
  enddo
!$omp end parallel

!$omp parallel do
doi = 1,10
  print *, 'Hello world', I
enddo
!$omp end parallel
```
Coarser-Grain Parallelism

What’s going on here? Is this possible? When?

Is this better? Why?

□ What’s going on here? Is this possible? When?
□ Is this better? Why?
**DO/FOR Directive: Clauses**

- `private (list)`
- `firstprivate(list)`
- `lastprivate(list)`
- `reduction(intrinsic operator : list)`
- `schedule(type[, chunk])`
- `ordered`
- `nowait`
**Parallel + for: Back to Motivation**

```c
omp_set_num_threads(2);
#pragma omp parallel private(i, j)
{
    #pragma omp for private (x,y)
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++) {
            x = i/ (double) m;
            y = j/ (double) n;
            depth[j][i] = mandel_val(x, y, maxiter);
        }
    #pragma omp for
    for (i = 1; i <= m; i++)
        for (j = 1; j <= n; j++)
            dith_depth[j][i] = 0.5*depth[j][i]
            + 0.25*(depth[j-1][i] + depth[j+1][i])
}
```
SECTIONS Directive

- **Fortran**
  ```fortran
  !$omp sections [clause [,] [clause ...]]
  !$omp section
    code for section 1
  !$omp section
    code for section 2
  ...
  !$omp end sections [nowait]
  ```

- **C/C++**
  ```c
  #pragma omp sections [clause [clause ...]]
  {
    [#pragma omp section]
      block
    ...
  }
  ```
SECTIONS Directive: Details

☐ Sections are assigned to threads
  ☐ Each section executes once
  ☐ Each thread executes zero or more sections
☐ Sections are not guaranteed to execute in any order

```c
#pragma omp parallel
#pragma omp sections
{
    X_calculation();
#pragma omp section
    y_calculation();
#pragma omp section
    z_calculation();
}
```
SECTIONS Directive: Clauses

- private (list)
- firstprivate(list)
- lastprivate(list)
- reduction(intrinsic operator : list)
- nowait
SINGLE Directive: Syntax

- **Fortran**
  
  ```fortran
  !$omp single [clause [,] [clause ...]]
  structured block
  !$omp end single [nowait]
  ```

- **C/C++**
  
  ```c
  #pragma omp single [clause [clause ...]]
  structured block
  ```

- **Clauses:**
  - `private (list)`
  - `firstprivate(list)`
  - `nowait`
Statements Between Loops

C$OMP PARALLEL DO
C$OMP& REDUCTION(+: sum)
do i=1,n
    sum = sum + a[i]
enddo
alpha = sum/scale
C$OMP PARALLEL DO
do i=1,n
    a[i] = alpha * a[i]
enddo

C$OMP PARALLEL
C$OMP DO REDUCTION(+: sum)
do i=1,n
    sum = sum + a[i]
enddo
C$OMP SINGLE
    alpha = sum/scale
C$OMP END SINGLE
C$OMP DO
    do i=1,n
        a[i] = alpha * a[i]
    enddo
C$OMP END PARALLEL
Restrictions on Work Sharing

- Structured blocks
  - Same nesting level
  - Restriction on branching as before
- If one thread reaches a work sharing construct, all threads reach the construct and in the same order
- Nesting of work sharing constructs is illegal
Parallel SECTIONS Directive

☐ Fortran

```fortran
!$omp parallel sections [clause [,] [clause ...]]
[$omp section]
   code for section 1
[$omp section]
   code for section 2]
...
!$omp end parallel sections [nowait]
```

☐ C/C++

```c
#pragma omp parallel sections [clause [clause ...]]
{
   [#pragma omp section]
   block
   ...
}
```
**Parallel Sections : Clauses**

- private(list)
- shared(list)
- firstprivate(list)
- lastprivate(list)
- default (private | shared | none)
- reduction(intrinsic operator : list)
- if(logical_expression)
- copyin(list)
Synchronization

- Concurrent access to shared data may result in data inconsistency
- Mechanism required to maintain data consistency
  - Mutual exclusion
- Sometimes code sections executed by different threads need to be sequenced in some particular order
  - Event synchronization
Mutual Exclusion

- Mechanisms for ensuring the consistency of data that is accessed concurrently by several threads
  - Critical Directive
  - Atomic Directive
  - Library Lock routines
Mutual Exclusion Features

- Apply to critical, atomic as well as library routines
  - NO Fairness guarantee
  - Guarantee of Progress
  - Careful when nesting - lots of chances for deadlock
Event Synchronization

- Mechanisms for controlling the relative order in which threads execute a section of code
  - Barriers
  - Ordered Sections
  - Master directive
Critical Section Directive

- **Fortran**
  ```fortran
  !$omp critical [(name)]
  structured block
  !$omp end critical [(name)]
  ```

- **C/C++**
  ```c
  #pragma omp critical [(name)]
  structured block
  ```
**Example: Critical Directive**

```c
cur_max = MINUS_INFINITY
!
$omp parallel do
    do i = 1, n
    ...
$omp critical
    if (a(i) .gt. cur_max) then
        cur_max = a(i)
    endif
$omp end critical
    ...
enddo
```
OpenMP 3.0

- Task parallelism is the big news!
  - Allows to parallelize irregular problems
    - unbounded loops
    - recursive algorithms
    - producer/consumer

- Loop parallelism Improvements
  - STATIC schedule guarantees
  - Loop collapsing
  - New AUTO schedule

- OMP_STACK_SIZE environment variable
General Task Characteristics in OpenMP 3.0

- A task has
  - Code to execute
  - A data environment (it owns its data)
  - An assigned thread that executes the code and uses the data

- Two activities: packaging and execution
  - Each encountering thread packages a new instance of a task (code and data)
  - Some thread in the team executes the task at some later time
## Definitions

- **Task construct**
  - Task directive plus structured block

- **Task**
  - The package of code and instructions for allocating data created when a thread encounters a task construct

- **Task region**
  - The dynamic sequence of instructions produced by the execution of a task by a thread
Task Construct

#pragma omp task [clause[[,]clause] ...]

structured-bloc

where clause can be one of:

if (expression)
untied
shared (list)
private (list)
firstprivate (list)
default( shared | none )
Intel Thread Building Blocks (TBB)

  - Main site: http://threadingbuildingblocks.org/
- C++ Library
- Targets threading for performance (designed to parallelize computationally intensive work)
- Is compatible with other threading packages
- Emphasizes scalable data parallel programming
- Relies on generic programming
- Specifies templates and tasks instead of threads
  - library schedules tasks onto threads
  - manages load balancing
Parallel_for

- parallel_for is a template function provided by library
- Example:
  - concurrently apply a function to each element in an array
- Serial version:
  ```c
  void SerialApplyFoo( float a[], size_t n ) {
    for( size_t i=0; i<n; ++i )
      Foo(a[i]);
  }
  ```
- Iteration space is 0…(n-1)
Parallel_for (continued)

- Parallel version requires two steps – Step 1:
  ```cpp
  #include "tbb_blocked_range.h"

  class ApplyFoo {
    float *const my_a;
  public:
    ApplyFoo( float a[] ) : my_a(a) {}
    void operator()( const blocked_range<size_t>& r ) const {
      float *a = my_a;
      for( size_t i=r.begin(); i!=r.end(); ++i )
        Foo(a[i]);
    }
  };
  ```
Parallel_for (continued)

- Parallel version requires two steps – Step 2:
  
  ```
  #include "tbb/parallel_for.h"
  
  void ParallelApplyFoo( float a[], size_t n ) {
    parallel_for(blocked_range<size_t>(0,n,IdealGrainSize),
                  ApplyFoo(a));
  }
  
  parallel_for breaks iteration space into chunks each of which are run on separate threads
  
  blocked_range<T>(begin,end,grainsize) - recursively divisible struct
  
  blocked_range is for 1D iteration spaces
  
  - TBB provides 2D and users can define their own
  
  - grainsize specifies the number of iterations for a “reasonable size” chunk to deal out to a processor
    
    - If the iteration space has more than grainsize iterations, parallel_for splits it into separate subranges that are scheduled separately
  
  - operator () processes a chunk
TBB Library - Algorithms

- parallel_reduce
  - math ops with elements of an array in parallel
- parallel_do
  - loops indeterminate length iteration spaces
- parallel_*
  - several others…
TBB Library – Pipeline

- TBB implements the pipeline pattern
- Data flows through a series of pipeline stages, and each stage processes the data in some way
- Given an incoming stream of data, some stages operate in parallel and others do not
**TBB – Containers**

- `concurrent_hash_map<Key, T, HashCompare>`
- `concurrent_queue<T, Allocator>`
- `concurrent_vector<T, Allocator>`
- Supports concurrent access, concurrent operations and parallel iteration
- TBB library retains control over memory allocation
TBB – Allocation

- `tbb Allocator<T>`
  - Allocates and frees memory via the TBB malloc library if available, otherwise it reverts to using malloc and free

- `scalable Allocator<T>`
  - Allocates and frees memory in a way that scales with the number of processors

- `others…`
TBB – Synchronization

- Library provides several flavors of mutexes including: recursive, spin, queuing, reader/writer...
- `atomic<T>`
  - read, write
  - fetch-and-add, fetch-and-store, compare-and-swap
Next Class

- Parallel algorithms