Programming Shared Address Space Platforms using OpenMP

Jinkyu Jeong (jinkyu@skku.edu)
Computer Systems Laboratory
Sungkyunkwan University
http://csl.skku.edu
Announcement

• Midterm exam
  – When: 4/24 (Mon) 10:30~11:45
  – Where: #400112 (Semiconductor Bldg.)
  – Stay tuned on our class web page
  – http://csl.skku.edu/SSE3054S17/News

• No class on 4/19 (Wed)

• Make-up class on 4/26 (Wed)
Topic Overview

• Introduction to OpenMP

• OpenMP directives
  – Concurrency control
    • parallel, for, sections
  – Synchronization
    • reduction, barrier, single, master, critical, atomic, ordered, …
  – Data handling
    • private, shared, firstprivate, lastprivate, threadprivate, …

• OpenMP library APIs

• Environment variables
OpenMP

• **Open** specifications for **Multi Processing**
• A standard for directive-based Parallel Programming
  – Shared-address space programming
  – FORTRAN, C, and C++
  – Support concurrency, synchronization, and data handling
  – Obviate the need for explicitly setting up mutexes, condition variables, data scope, and initialization
OpenMP Solution Stack

User Application Layer

OpenMP Program Layer
- Directives, Compiler
- OpenMP Library
- Environment Variables

System Layer
- Runtime Library
- OS/System support with Shared memory
Parallel Programming Practice

• Current
  – Start with a parallel algorithm
  – Implement, keeping in mind
    • Data races
    • Synchronization
    • Threading syntax
  – Test & Debug
  – Debug ….

• Ideal way
  – Start with some algorithm
  – Implement serially, ignoring
    • Data races
    • Synchronization
    • Threading syntax
  – Test & Debug
  – Auto-magically parallelize
Implementation on Shared Memory

- **Thread Library**
  - Library calls
  - Low level programming
    - Explicit thread creation & work assignment
    - Explicit handling of synchronization
  - Parallelism expression
    - Task: create/join thread
    - Data: detailed programming
  - Design concurrent version from the start

- **OpenMP**
  - Compiler directives
  - Higher abstraction
    - Compilers convert code to use OpenMP library, which is actually implemented with thread APIs
  - Parallelism expression
    - Task: task/taskwait, parallel sections
    - Data: parallel for
  - Incremental development
    - Start with sequential version
    - Insert necessary directives
Implementation Examples

• Threaded functions
  – Exploit data parallelism

```c
node A[N], B[N];

main() {
    for (i=0; i<nproc; i++)
        thread_create(par_distance);
    for (i=0; i<nproc; i++)
        thread_join();
}
void par_distance() {
    tid = thread_id();
    n = ceiling(N/nproc);
    s = tid * n;
    e = MIN((tid+1)*n, N);
    for (i=s; i<e; i++)
        for (j=0; j<N; j++)
            C[i][j] = distance(A[i], B[j]);
}
```

• Parallel loops
  – Exploit data parallelism

```c
node A[N], B[N];

#pragma omp parallel for
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        C[i][j] = distance(A[i], B[j]);
```
OpenMP Programming Model

• **Fork-join model**
  – Thread pool
  – Implicit barrier

  – `#pragma omp`
    • `parallel for`
    • `parallel sections`

• **Data scoping semantics are somewhat complicated**
  – private, shared, copyin, firstprivate, lastprivate, copyprivate, threadprivate, …
  – Implicit rules,…
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

void Hello(void); /* Thread function */

int main(int argc, char* argv[]) {
    /* Get number of threads from command line */
    int thread_count = strtol(argv[1], NULL, 10);

    #pragma omp parallel num_threads(thread_count)
    Hello();

    return 0;
} /* main */

void Hello(void) {
    int my_rank = omp_get_thread_num();
    int thread_count = omp_get_num_threads();

    printf("Hello from thread %d of %d\n", my_rank, thread_count);
} /* Hello */
```
Hello World in OpenMP – con’t

• Compile
  – #gcc -g -Wall -fopenmp -o omp_hello omp_hello.c

• Run
  – #./omp_hello 4

Possible outcomes

Hello from thread 0 of 4
Hello from thread 1 of 4
Hello from thread 2 of 4
Hello from thread 3 of 4
Pragmas

• Special compiler directives
  – `#pragma`
  – Provides extension to the basic C (or C++)
  – Compilers that don’t support the pragmas ignore them

• OpenMP pragmas
  `#pragma omp directive [clause list]` 
  `/* structured block */`
  – Directives specify actions OpenMP supports
  – Additional clauses follow the directive
  – Parallel directive
    `#pragma omp parallel [clause list]`
  • Most basic parallel directive in OpenMP
parallel Directive

#pragma omp parallel [clause list]
/* structured block */

• Possible clauses
  – Conditional Parallelization
    • if (scalar expression)
      – Determines whether to create threads or not
  – Degree of Concurrency
    • num_threads (integer expression)
      – Specifies the number of threads that are created.
  – Data Handling
    • private (variable list)
      – Variables local to each thread
    • firstprivate (variable list)
      – Variables are initialized to corresponding values before the parallel directive
    • shared (variable list)
      – Variables are shared across all the threads.
    • default (shared|private|none)
      – Default data handling specifier
Example of parallel Directive

```c
#pragma omp parallel if (is_parallel==1) num_threads(8) \
    private (a) shared (b) firstprivate(c) default(none){
    /* structured block */
}
```

• **if** (is_parallel==1) **num_threads** (8)
  – If the value of the variable is_parallel equals one, eight threads are created.

• **private** (a)
  – Threads get private copy of variable a

• **firstprivate** (c)
  – Threads get private copy of variable c
  – The value of each copy of c is initialized to the value of c before the parallel directive.

• **shared** (b)
  – Threads share a single copy of variable b.

• **default** (none)
  – Default scope of variables are none
  – Compile error when not all variables are specified as shared or private
reduction Clause

- **A reduction**
  - Applies the same reduction operator to a sequence of operands to get a single result
  - All of the intermediate results of the operation should be stored in the same variable: the reduction variable

- **Reduction clause in OpenMP**
  - `reduction(<operator>: <variable list>)`
  - `#pragma omp parallel reduction(+: sum) num_threads(8) {
    /* compute local sums here */
  }
  - The variables in the list are implicitly specified as being private to threads.
  - Reduction operators ⇒ `+`, `*`, `-`, `&`, `|`, `^`, `&&`, `||`
  - Commutative and associative operators can provide correct results
Example: estimating Pi

- Estimating Pi using Monte Carlo method

Image credit: wikipedia.org
Estimating Pi using OpenMP

/* ******************************************************
An OpenMP version of a threaded program to compute PI.
****************************************************** */

#pragma omp parallel default(private) shared(npoints) \ 
  reduction(+: sum) num_threads(8)
{
  num_threads = omp_get_num_threads();
  sample_points_per_thread = npoints / num_threads;
  sum = 0;
  for (i = 0; i < sample_points_per_thread; i++) {
    rand_no_x = (double)rand_r(&seed)/RAND_MAX;
    rand_no_y = (double)rand_r(&seed)/RAND_MAX;
    if ( x * x + y * y < 1.0)
      sum ++;
  }
}
Estimating Pi using Pthreads (1)

```c
#include <pthread.h>
#include <stdlib.h>
#define MAX_THREADS 512
void *compute_pi (void *);
....
main() {
    ...
pthread_t p_threads[MAX_THREADS];
pthread_attr_t attr;
pthread_attr_init (&attr);
for (i=0; i< num_threads; i++) {
    hits[i] = i;
    pthread_create(&p_threads[i], &attr, compute_pi, 
    (void *) &hits[i]);
}
for (i=0; i< num_threads; i++) {
    pthread_join(p_threads[i], NULL);
    total_hits += hits[i];
}
...}
```
void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x, y;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {
        x = (double)rand_r(&seed)/RAND_MAX;
        y = (double)rand_r(&seed)/RAND_MAX;
        if ( x * x + y * y < 1.0 )
            local_hits ++;
        seed *= i;
    }
    *hit_pointer = local_hits;
    pthread_exit(0);
}
for Directive

• Split parallel iteration spaces (i.e., loop) across threads
• Implicit barrier at the end of a loop
• General form

    #pragma omp for [clause list]
    /* for loop */

• Possible clauses
  – private, firstprivate, lastprivate, reduction, schedule, nowait, and ordered.
Example: Estimating Pi using for

```c
#pragma omp parallel default(private) \ 
    shared(npoints) reduction(+: sum) num_threads(8) 
{
    sum = 0;
    #pragma omp for
    for (i = 0; i < sample_points_per_thread; i++) {
        rand_no_x = (double)rand_r(&seed)/RAND_MAX;
        rand_no_y = (double)rand_r(&seed)/RAND_MAX;
        if ( x * x + y * y < 1.0)
            sum ++;
    }
}
```
Assigning Iterations to Threads

- **parallel for** directive
  - Basic partitioning policy → block partitioning

<table>
<thead>
<tr>
<th>Iteration space</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Is this optimal?
  - **Yes**, when each iteration takes equal time
  - **No**, when each iteration takes different time
    - ex) larger index takes longer time
Assigning Iterations to Threads

• Example of $f(i)$
  – $f(i)$ calls sin function $i$ times
  – Time($f(i)$) is linear to $i$

• Block partitioning

```java
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;
    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
} /* f */
```

• Cyclic partitioning
Assigning Iterations to Threads

• Example of $f(i)$
  – $f(i)$ calls sin function $i$ times
  – Time($f(i)$) is linear to $i$
  – $n = 10,000$

```java
double f(int i) {
    int j, start = i*(i+1)/2, finish = start + i;
    double return_val = 0.0;
    for (j = start; j <= finish; j++) {
        return_val += sin(j);
    }
    return return_val;
}
```

<table>
<thead>
<tr>
<th></th>
<th>One thread</th>
<th>Two threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Block</td>
<td>Cyclic</td>
</tr>
<tr>
<td>Run-time</td>
<td>3.67s</td>
<td>2.77s</td>
</tr>
<tr>
<td>Speed-up</td>
<td>1x</td>
<td>1.33x</td>
</tr>
</tbody>
</table>

• Scheduling loop (load balancing) is important
Schedule Clause

- **Format**
  - `#pragma omp parallel for schedule(type [, chunk])`
    - `type` := static, dynamic, guided, runtime
    - `chunk` := positive integer
  - **static**
    - Divide iterations by `chunk` (near equal in size by default)
    - Statically assign threads in a round-robin fashion
  - **dynamic**
    - Divide iterations by `chunk` (1 by default)
    - Dynamically assign a chunk to an idle thread (master/worker)
  - **guided**
    - Chunk size is reduced in an exponentially decreasing manner
    - Dynamically assign a chunk to an idle thread (master/worker)
    - Minimum chunk size is specified by `chunk` (1 by default)
  - **runtime**
    - Determined at runtime with OMP_SCHEDULE environment variable
Schedule Clause (Illustration)

• Dividing iteration space
  – **Static** schedule on iteration space
  
  
  \[ 0 \quad \frac{1}{4}N \quad \frac{1}{2}N \quad \frac{3}{4}N \quad N-1 \]

  – **Dynamic** schedule on iteration space (master/worker)

  – **Guided** schedule on iteration space (master/worker)
The Static Schedule Type

• Static schedule on iteration space

  \[
  0 \quad \frac{1}{4}N \quad \frac{1}{2}N \quad \frac{3}{4}N \quad N-1
  \]

• Example
  – 12 iterations (0, 1, \ldots, 11) and 3 threads

<table>
<thead>
<tr>
<th>schedule (static, 1)</th>
<th>schedule (static, 2)</th>
<th>schedule (static, 4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread 0 : 0, 3, 6, 9</td>
<td>Thread 0 : 0, 1, 6, 7</td>
<td>Thread 0 : 0, 1, 2, 3</td>
</tr>
<tr>
<td>Thread 1 : 1, 4, 7, 10</td>
<td>Thread 1 : 2, 3, 8, 9</td>
<td>Thread 1 : 4, 5, 6, 7</td>
</tr>
<tr>
<td>Thread 2 : 2, 5, 8, 11</td>
<td>Thread 2 : 4, 5, 10, 11</td>
<td>Thread 2 : 8, 9, 10, 11</td>
</tr>
</tbody>
</table>
The Dynamic Schedule Type

#pragma omp parallel for schedule(dynamic, \ chunksize)

- **Dynamic** schedule on iteration space (master/worker)
  
  - The iterations are broken up into chunks of **chunksize** consecutive iterations
  - Each thread executes a chunk
  - When a thread finishes a chunk, it requests another one from the run-time system
  - This continues until all the iterations are completed
  - The **chunksize** is 1 by default
The Guided Schedule Type

```c
#pragma omp parallel for schedule(guided, \chunksize)
```

- **Guided** schedule on iteration space (master/worker)

  - Initial size of chunk is **\# of iterations / \# of threads**
  - Each thread executes a chunk, and when a thread finishes a chunk, it requests another one from runtime system
  - The **size of the new chunk decreases exponentially**
  - If no **chunksize** is specified, the size of the chunks decreases down to 1
  - If **chunksize** is specified, it decreases down to **chunksize**
    - The very last chunk can be smaller than **chunksize**
nowait Clause

• Worksharing for loops end with an implicit barrier
• Often, less synchronization is appropriate
  – When a series of for directives are in a parallel construct
• nowait clause
  – Used with a for directive
  – Avoids implicit barrier at the end of for

```c
#pragma omp for
for ( ... )
#pragma omp for
for ( ... )
#pragma omp for nowait
for ( ... )
#pragma omp for
for ( ... )
```
sections Directive

- Non-iterative parallel task assignment using the sections directive.

- General form

  ```
  #pragma omp sections [clause list]
  {
    #pragma omp section
    /* structured block */
    #pragma omp section
    /* structured block */
    ...
  }
  ```

- Possible clauses
  - nowait, shared, private, ...

- Implicit barrier at the end of sections
sections Directive: Example

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```

Diagram:

- For `n` threads available:
  - Task `A()` executed by first thread
  - Task `B()` executed by second thread
  - Task `C()` executed by third thread

- For 1 thread available:
  - Task `A()` executed by thread
  - Task `B()` executed by thread
  - Task `C()` executed by thread

SSE3054: Multicore Systems, Spring 2017, Jinkyu Jeong (jinkyu@skku.edu)
Merging Directives (I)

```c
#pragma omp parallel
{
    #pragma omp sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            taskA();
        }
        #pragma omp section
        {
            taskB();
        }
        #pragma omp section
        {
            taskC();
        }
    }
}
```
Merging Directives (2)

```
#pragma omp parallel
{
    #pragma omp for
    for (i = 0; i < mmax; i++)
        /* body of loop */
}

#pragma omp parallel for
{
    for (i = 0; i < mmax; i++)
        /* body of loop */
}
```
Caution for Merging Directives (1)

- Each parallel directive forks threads
- Then, join threads after the parallel construct

```c
#pragma omp parallel for
for (i=0; i<n; ++i) {
    ...
}
#pragma omp parallel for
for (i=0; i<n; ++i) {
    ...
}
```

Diagram:
- First loop
- Second loop
- Fork
- Join
Caution for Merging Directives (2)

- Parallelize a loop using threads that are forked in advance

```c
#pragma omp parallel num_threads(n)
{
    #pragma omp for
    for (i=0; i<n; ++i) {
        ...
    }
    #pragma omp for
    for (i=0; i<n; ++i) {
        ...
    }
}
```

fork threads for the following structured block
parallelize the following for loop using the pre-forked threads

Unnecessary fork&join is eliminated
Nesting parallel Directives

• Nested parallelism can be enabled using the OMP_NESTED environment variable.
  – If the OMP_NESTED environment variable is set to TRUE, nested parallelism is enabled.
  – In this case, each parallel directive creates a new team of threads.
Synchronization Constructs in OpenMP

• OpenMP provides a variety of synchronization constructs:

  #pragma omp barrier
  #pragma omp single [clause list]
    structured block
  #pragma omp master
    structured block
  #pragma omp critical [(name)]
    structured block
  #pragma omp ordered
    structured block
barrier Directive

• OpenMP
  – Barrier synchronization
    • Wait until all the threads in a team reach to the point
  – #pragma omp barrier

```c
main() {
    #pragma omp parallel
    sub();
}
sub() {
    work1();
    #pragma omp barrier
    work2();
}
```
**single Directive**

- Executed by one thread within a parallel region
  - Any thread can execute the single region
  - Implicit barrier synchronization at the end

```c
#pragma omp parallel
{
    #pragma omp single
    {
        a = 10;
    } /* implicit barrier */

    #pragma omp for
    for (i=0; i<N; i++)
        B[i] = a;
}
/* end of parallel region */
```
master Directive

• Executed by the master thread
  – No implicit barrier
  – If a barrier is needed for correctness, it must be specified

```c
#pragma omp parallel
{
  #pragma omp master
  {
    a = 10;
  } /* no barrier */
  #pragma omp barrier

  #pragma omp for
  for (i=0; i<N; i++)
    B[i] = a;

} /* end of parallel region */
```
critical Directive

• Format
  – # pragma omp critical
  – Provides mutual exclusion of the following structured block to all threads in a team
    
    # pragma omp critical
    Enqueue(&queue, &message)
    # pragma omp critical
    Dequeue(&queue, &message)

• Limitation
  – Distinct critical sections are treated as one composite critical section
  – Serialization of all threads

Critical sections for queue

Critical sections for stack

# pragma omp critical
Enqueue(&queue, &message)
# pragma omp critical
Dequeue(&queue, &message)

# pragma omp critical
Push(&stack, &message)
# pragma omp critical
Pop(&stack, &message)
Named critical Directive

• Format
  – # pragma omp critical (name)
  – Specifies the name of a critical section
  – OpenMP provides mutual exclusion to the critical sections having the same name

```
# pragma omp critical(queue)
Enqueue(&queue, &message)

# pragma omp critical(stack)
Push(&stack, &message)
```

```
# pragma omp critical(queue)
Dequeue(&queue, &message)

# pragma omp critical(stack)
Pop(&stack, &message)
```

• Limitation
  – Distinction of critical sections is made at compilation time
  – No critical section distinction between different data structures
Lock APIs in OpenMP

• When to distinct critical sections based on the data structure

• Usage
  – `omp_lock_t lock;`
  – `omp_init_lock(&lock); omp_destroy_lock(&lock);`
  – `omp_set_lock(&lock); omp_unset_lock(&lock);`

• Example

```c
/* q_p = msg_queues[dest] */
omp_set_lock(&q_p->lock);
Enqueue(q_p, my_rank, msg);
omp_unset_lock(&q_p->lock);

/* q_p = msg_queues[my_rank] */
omp_set_lock(&q_p->lock);
Dequeue(q_p, &src, &msg);
omp_unset_lock(&q_p->lock);
```
atomic Directive

• Format
  – #pragma omp atomic

• It only protects critical sections that consist of a single C assignment statement

• Valid statement format:

  x <op>= <expression>;
  x++;         
  ++x;         
  x--;         
  --x;

• Supported operations:

  +, *, -, /, &, ^, |, <<, or >>
#pragma omp parallel for ordered
private(i) shared(a, b)
{
  for (i = 0; i < mmax; i++)
  {
    /* other processing on b[i] */
    #pragma omp ordered
    b[i] = b[i-1] + a[i]
  }
}
Data Handling Clauses

• lastprivate
  – The last value of a variable is kept after join of threads

• threadprivate
  – Each thread has a local copy of a variable similar to private
  – But, the variable is alive across different parallel constructs

• copyin
  – Initialize a threadprivate variable from the value of variable in a master thread
OpenMP Programming Model

- Task model (OpenMP 3.0 – released, May 2008)
  - Task creation and join
  - Can handle
    - Unbounded loops
    - Recursive algorithms
    - Producer/consumer
  - `#pragma omp task [clause list]`
    - `task`
    - `taskwait`
  - (NOTE) parallel sections use fork-join model
    - Not suitable for above mentioned jobs
Example: OpenMP Task

- Task level parallelism

```c
void traverse (NODE *p) {
    if (p->left)
        traverse(p->left);
    if (p->right)
        traverse(p->right);
    process(p);
}
```

```c
void traverse (NODE *p) {
    if (p->left)
        #pragma omp task
        traverse(p->left);
    if (p->right)
        #pragma omp task
        traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

- Post-order visit
- Individual join?
- Join all the descendant tasks
  - Join all the task created so far
  - Taskgroup is needed (Not defined in OpenMP 3.0)
Example: Linked List Traversal

```c
while(my_pointer) {
    do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
```
Example: Linked List Traversal

```c
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next ;
        }
    } // End of single
} // End of parallel region
```
Example: Linked List Traversal

```c
my_pointer = listhead;
#pragma omp parallel
{
    #pragma omp single nowait
    {
        while(my_pointer) {
            #pragma omp task firstprivate(my_pointer)
            {
                do_independent_work (my_pointer);
            }
            my_pointer = my_pointer->next;
        }
    } // End of single — no implied barrier
} // End of parallel region — implicit barrier
```
OpenMP Library Functions

• Control the execution of threaded programs.
  – `void omp_set_num_threads (int num_threads);`
    • Set max # of threads for next parallel construct
  – `int omp_get_num_threads ();`
    • Get active # of threads
  – `int omp_get_max_threads ();`
    • Get maximum # of threads
  – `int omp_get_thread_num ();`
    • Return thread ID (from 0 to MAX-1)
  – `int omp_get_num_procs ();`
    • Get # of processors available
  – `int omp_in_parallel();`
    • Determines whether running in parallel construct
OpenMP Library Functions

• Controlling and monitoring thread creation
  – `void omp_set_dynamic (int dynamic_threads);`
    • Enable dynamic change of # of threads for parallel construct
  – `int omp_get_dynamic ();`
    • Query
  – `void omp_set_nested (int nested);`
    • Enable nested parallel directive
  – `int omp_get_nested ();`
    • Query
Environment Variables in OpenMP

- **OMP_NUM_THREADS**
  - Specifies the default number of threads created upon entering a parallel region.

- **OMP_SET_DYNAMIC**
  - Determines if the number of threads can be dynamically changed.

- **OMP_NESTED**
  - Turns on nested parallelism.

- **OMP_SCHEDULE**
  - Scheduling of for-loops if the clause specifies runtime
  - Example
    
    ```
    $ export OMP_SCHEDULE="static, 1"
    $ ./omp_program 4
    static scheduling with chunksize of 1
    ```
OpenMP Programming Practice

• OpenMP
  – Start with a parallelizable algorithm
  – Implement serially, mostly ignoring
    • Data races
    • Synchronization
    • Threading syntax
  – Test & Debug
  – Annotation with directives for parallelization & synchronization
  – Test & Debug

• Ideal way
  – Start with some algorithm
  – Implement serially, ignoring
    • Data races
    • Synchronization
    • Threading syntax
  – Test & Debug
  – Auto-magically parallelize
OpenMP Summary

• **OpenMP is:**
  – An API that may be used to explicitly direct multi-threaded, shared memory parallelism
  – Portable
    • C/C++ and Fortran support
    • Implemented on most Unix variants and Windows
  – Standardized
    • Major computer HW and SW vendors jointly defines (OpenMP.org)

• **OpenMP does NOT:**
  – Support distributed memory systems
    • but Cluster OpenMP does
  – Automatically parallelize
  – Have data distribution controls
  – Guarantee efficiency, freedom from data races, …
References

• “Introduction to Parallel Computing”, by Ananth Grama, Anshul Gupta, George Karypis, and Vipin Kumar, Addison Wesley, 2003

• “COMP422: Parallel Computing” by Prof. John Mellor-Crummey at Rice Univ.

• "OpenMP Tasking Explained," Ruud van der Pas, SC 13