ME964
High Performance Computing for Engineering Applications

Parallel Computing using OpenMP
[Part 2 of 2]
April 5, 2011

“The inside of a computer is as dumb as hell but it goes like mad!”
Richard Feynman
Before We Get Started...

- Last time
  - General intro, OpenMP
  - Parallel regions
  - Work sharing under OpenMP
    - omp for
    - omp sections
    - omp tasks

- Today
  - Parallel Computing using OpenMP, part 2 of 2.

- Other issues
  - Assignment 7 due on April 7
  - Thursday I’ll finish what I planned to lecture for ME964
  - Beyond that:
    - No class next Tuesday
    - Recall that you have to send me a PPT with your Final Project topic (see syllabus for due date)
    - We’ll have several guest lecturers
    - Midterm Exam on April 19
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p) {
            //block 2
            #pragma omp task private(p)
            process(p);
            p = p->next;  //block 3
        }
    }
}
```

Includes material from IOMPP
Tasks: Putting Things in Perspective

- Upper pic: sequential. Lower pic: parallel

Credit: Wikipedia
Tasks: Synchronization Issues

- Setup:
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guaranteed completion of Task A before invoking the execution of Task B

- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo() tasks created here – one for each thread
- All foo() tasks guaranteed to be completed here
- One bar() task created here
- bar() task guaranteed to be completed here

Credit: IOMPP
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization
- Advanced topics
Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model

- **Shared variable** - a variable that can be read or written by multiple threads

- Shared clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
  - Other examples of variables being shared among threads
    - File scope variables
    - Namespace scope variables
    - Variables with const-qualified type having no mutable member
    - Static variables which are declared in a scope inside the construct

Includes material from IOMPP
Data Scoping – What’s Private

- Not everything is shared...

- Examples of implicitly determined PRIVATE variables:
  - Stack (local) variables in functions called from parallel regions
  - Automatic variables within a statement block
  - Loop iteration variables
  - Implicitly declared private variables within tasks will be treated as firstprivate

- **firstprivate**
  - Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct

Includes material from IOMPP
A Data Environment Example

float A[10];
main () {
    int index[10];
    #pragma omp parallel
    {
        Work (index);
    }
    printf ("%d\n", index[1]);
}

extern float A[10];
void Work (int *index)
{
    float temp[10];
    static integer count;
    <...>
}

A, index, count are shared by all threads, but temp is local to each thread

Includes material from IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task
    x = fib(n-1);
    #pragma omp task
    y = fib(n-2);
    #pragma omp taskwait
    return x+y
}
```

- n is private in both tasks
- x is a private variable
- y is a private variable

Values of the private variables not available outside of tasks

Credit: IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait
    return x+y
}
```

- `n` is private in both tasks
- `x` & `y` are shared

Good solution: we need both values to compute the sum

The values of the `x` & `y` variables will be available outside each task construct – after the `taskwait`
Discussion: Variable Scoping Aspects

- Consider parallelizing the following code

```c
int main() {
    const int n=20;
    int a[n];
    for( int i=0; i<n; i++ )
        a[i] = i;

    //this is the part that needs to be parallelized
    caller(a, n);

    for( int i=0; i<n; i++ )
        printf("a[%d]=%d\n", i, a[i]);
    return 0;
}

void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output

- Looks good
  - The value of the counter increases each time you hit the “callee” subroutine

- If you run the executable 20 times, you get the same results 20 times
First Attempt to Parallelize

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Constant decl. outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>(actually a)</td>
</tr>
<tr>
<td>y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>(actually k)</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>(actually j)</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cv</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Program Output, First Attempt to Parallelize

- Looks bad…
  - The values in array “a” are all over the map
  - The value of the counter “cv” changes chaotically within “callee”
  - The function “callee” gets hit a random number of times (should be hit 100 times). Example:
    
    ```
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 70
    ```

- If you run executable 20 times, you get different results

- One of the problems is that “j” is shared
Second Attempt to Parallelize

- Declare the inner loop variable “j” as a private variable within the parallel loop

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Second Attempt to Parallelize

- Looks better
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)
  - Fortunately, it’s not used in this code for any subsequent computation

- Conclusion: be careful when you work with static or some other global variables in parallel programming
  - In general, dealing with such variables is bad programming practice
Slightly Better Solution…

- Declare the inner loop index “j” only inside the parallel segment
  - After all, it’s only used there
  - You get rid of the “private” attribute, less constraints on the code, increasing the opportunity for code optimization at compile time

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

Used here, then you should declare here (common sense…)

Program Output, Parallelized Code

- Looks good
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)

- What surprised me: the value of the counter was indeed 100
  - In other words, although shared, no trashing of this variable…
Work Plan

What is OpenMP?
- Parallel regions
- Work sharing
- Data environment

Synchronization

- Advanced topics
Implicit Barriers

- Several OpenMP constructs have implicit barriers
  - parallel – necessary barrier – cannot be removed
  - for
  - single

- Unnecessary barriers hurt performance and can be removed with the `nowait` clause
  - The `nowait` clause is applicable to:
    - for clause
    - single clause
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```c
#pragma omp for nowait
for(...)
{
...;
}
```

```c
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);
```

```c
#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```

Credit: IOMPP
Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
    #pragma omp barrier
    DoSomeWork(B,C); // Processed B into C
}
```

Credit: IOMPP
Atomic Construct

- Applies only to simple update of memory location
- Special case of a critical section, to be discussed shortly

```c
#pragma omp parallel for shared(x, y, index, n)
for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;

Credit: IOMPP
Example: Dot Product

```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?

Credit: IOMPP
Race Condition

- A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable.

- For example, suppose both Thread A and Thread B are executing the statement

  ```
  area += 4.0 / (1.0 + x*x);
  ```

Credit: IOMPP
## Two Possible Scenarios

<table>
<thead>
<tr>
<th>Value of area</th>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>11.667</td>
<td>+3.765</td>
<td>15.432</td>
</tr>
<tr>
<td>15.432</td>
<td></td>
<td>15.432</td>
</tr>
<tr>
<td>18.995</td>
<td></td>
<td>+3.563</td>
</tr>
</tbody>
</table>

Order of thread execution causes non deterministic behavior in a data race.

Credit: IOMPP
Protect Shared Data

- Must protect access to shared, modifiable data

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical
        sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

```
#define pragma omp critical [(lock_name)]
```

- Defines a critical region on a structured block

Threads wait their turn – only one at a time calls consum() thereby protecting RES from race conditions.

Naming the critical construct RES_lock is optional but highly recommended.

```c
float RES;
#pragma omp parallel
{
#pragma omp for
  for(int i=0; i<niters; i++){
    float B = big_job(i);
    #pragma omp critical (RES_lock)
    consum(B, RES);
  }
}
```

Good Practice – Name all critical sections

Includes material from IOMPP
OpenMP Reduction Clause

\[ \text{reduction (op : list)} \]

- The variables in “list” must be shared in the enclosing parallel region.

- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”.
  - These copies are updated locally by threads.
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable.

Credit: IOMPP
Reduction Example

- Local copy of `sum` for each thread
- All local copies of `sum` added together and stored in “global” variable

```c
#pragma omp parallel for reduction(+:sum)
for(i=0; i<N; i++) {
    sum += a[i] * b[i];
}
```
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi \]

```c
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
```

Credit: IOMPP
OpenMP Reduction Example: Numerical Integration

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1./(double(num_steps));
    double sum;

    #pragma omp parallel for reduction(+:sum)
    {
        for(int i=0; i<num_steps; i++) {
            double x = (i + .5)*step;
            sum += 4.0/(1.+ x*x);
        }
    }

    double my_pi = sum*step;

    return 0;
}

This didn’t work for me in VS2008, no support for reduction there…
```
C/C++ Reduction Operations

- A range of associative operands can be used with reduction
- Initial values are the ones that make sense mathematically

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
</tbody>
</table>

Credit: IOMPP
OpenMP:
Concluding Remarks & Wrap up
OpenMP Summary

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997
  - OpenMP Architecture Review Board (ARB) determines additions and updates to standard
    - The draft of OpenMP Version 3.1 has been released for public comments on 02/07/2011
    - The final specification of Version 3.1 is expected for June 2011

Include material from Rebecca Hartman-Baker’s presentation
The OpenMP API

- Application Programmer Interface (API) is combination of
  - Directives
    - Example: `#pragma omp task`
  - Runtime library routines
    - Example: `int omp_get_thread_num(void)`
  - Environment variables
    - Example: `setenv OMP_SCHEDULE "guided, 4"`

Include material from Rebecca Hartman-Baker’s presentation
The OpenMP API

[Cntd.]

API falls into three categories

- Expression of parallelism (flow control)
  - Example: `#pragma omp parallel for`

- Data sharing among threads (communication)
  - Example: `#pragma omp parallel for private(x,y)`

- Synchronization (coordination or interaction)
  - Example: `#pragma omp barrier`

Include material from Rebecca Hartman-Baker’s presentation
OpenMP: Environment Variables

- **OMP_SCHEDULE**
  - Example: `setenv OMP_SCHEDULE "guided, 4"

- **OMP_NUM_THREADS**
  - Sets the maximum number of threads to use during execution.
  - Example: `setenv OMP_NUM_THREADS 8`

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE
  - Example: `setenv OMP_DYNAMIC TRUE`

- **OMP_NESTED**
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE
  - Example: `setenv OMP_NESTED TRUE`
OpenMP: Environment Variables

[New ones in 3.0 Release]

- **OMP_STACKSIZE**
  - Controls the size of the stack for created (non-Master) threads.

- **OMP_WAIT_POLICY**
  - Provides a hint to an OpenMP implementation about the desired behavior of waiting threads.

- **OMP_MAX_ACTIVE_LEVELS**
  - Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
    - `setenv OMP_MAX_ACTIVE_LEVELS 2`

- **OMP_THREAD_LIMIT**
  - Sets the number of OpenMP threads to use for the whole OpenMP program
  Example:
    - `setenv OMP_THREAD_LIMIT 8`
# OpenMP 3.0:
## Summary of Run-Time Library OpenMP Routines

<table>
<thead>
<tr>
<th>Routine</th>
<th>Routine</th>
</tr>
</thead>
<tbody>
<tr>
<td>OMP_SET_NUM_THREADS</td>
<td>OMP_GET_ANCESTOR_THREAD_NUM</td>
</tr>
<tr>
<td>OMP_GET_NUM_THREADS</td>
<td>OMP_GET_TEAM_SIZE</td>
</tr>
<tr>
<td>OMP_GET_MAX_THREADS</td>
<td>OMP_GET_ACTIVE_LEVEL</td>
</tr>
<tr>
<td>OMP_GET_THREAD_NUM</td>
<td>OMP_INIT_LOCK</td>
</tr>
<tr>
<td>OMP_GET_THREAD_LIMIT</td>
<td>OMP_DESTROY_LOCK</td>
</tr>
<tr>
<td>OMP_GET_NUM_PROCS</td>
<td>OMP_SET_LOCK</td>
</tr>
<tr>
<td>OMP_IN_PARALLEL</td>
<td>OMP_UNSET_LOCK</td>
</tr>
<tr>
<td>OMP_SET_DYNAMIC</td>
<td>OMP_TEST_LOCK</td>
</tr>
<tr>
<td>OMP_GET_DYNAMIC</td>
<td>OMP_INIT_NEST_LOCK</td>
</tr>
<tr>
<td>OMP_SET_NESTED</td>
<td>OMP_DESTROY_NEST_LOCK</td>
</tr>
<tr>
<td>OMP_GET_NESTED</td>
<td>OMP_SET_NEST_LOCK</td>
</tr>
<tr>
<td>OMP_SET_SCHEDULE</td>
<td>OMP_UNSET_NEST_LOCK</td>
</tr>
<tr>
<td>OMP_GET_SCHEDULE</td>
<td>OMP_TEST_NEST_LOCK</td>
</tr>
<tr>
<td>OMP_SET_MAX_ACTIVE_LEVELS</td>
<td>OMP_GET_WTIME</td>
</tr>
<tr>
<td>OMP_GET_MAX_ACTIVE_LEVELS</td>
<td>OMP_GET_WTICK</td>
</tr>
<tr>
<td>OMP_GET_LEVEL</td>
<td></td>
</tr>
</tbody>
</table>
30+ Library Routines

- Runtime environment routines:
  - Modify/check the number of threads
    `omp_[set|get]_num_threads()`
    `omp_get_thread_num()`
    `omp_get_max_threads()`
  - Are we in a parallel region?
    `omp_in_parallel()`
  - How many processors in the system?
    `omp_get_num_procs()`
  - Explicit locks
    `omp_[set|unset]_lock()`
OpenMP API

- Get the thread number within a team
  ```c
  int omp_get_thread_num(void);
  ```
- Get the number of threads in a team
  ```c
  int omp_get_num_threads(void);
  ```
- Usually not needed for OpenMP codes
  - Can lead to code not being serially consistent
  - Does have specific uses (debugging)
  - Must include a header file
    ```c
    #include <omp.h>
    ```
OpenMP
The 30,000 Feet Perspective

OpenMP language extensions

parallel control structures
- parallel directive
- do/parallel do and section directives
  - governs flow of control in the program

work sharing
- distributes work among threads

data environment
- scopes variables
  - shared and private clauses

synchronization
- coordinates thread execution
  - critical and atomic directives
  - barrier directive

runtime functions, env. variables
- runtime environment
  - omp_set_num_threads()
Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can implement complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, code is easy to read
- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives

Credit: Rebecca Hartman-Baker
OpenMP, Some Caveats

- I’m not familiar with various OpenMP distributions, but it seems that there is a lag caused by the vendors to support the latest specifications
  - Intel probably is most up to speed although I haven’t used their compilers

- OpenMP threads are heavy
  - Good for handling parallel tasks
  - Not so good at handling fine large scale grain parallelism
Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- OpenMP.org, http://openmp.org/
- OpenMP 3.0 API Summary Cards:
  - C/C++: