“The empires of the future are the empires of the mind.”

-- Winston Churchill
Before We Get Started…

- Last time: OpenMP
  - sections
  - tasks
  - data scoping
  - synchronization

- Today:
  - `reduce` operations in OpenMP
  - Closing comments on the OpenMP API

- Miscellaneous
  - Forum etiquette: please be nice to each other
  - HW posted online later today. Due next Mo, at 11:59 PM
  - Midterm Project: If you don’t hear from me, it means it’s ok
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?
Protect Shared Data

- The **critical** construct: protects access to shared, modifiable data
- The critical section allows only one thread to enter it at a given time

```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;
}
```
OpenMP Critical Construct

```
#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.

```
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);
  }
}
```

Includes material from IOMPP
OpenMP reduction Clause

reduction (op:list)

- The variables in list will be shared in the enclosing parallel region

- Here’s what happens inside the 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
Local copy of `sum` for each thread engaged in the reduction is private
- Each local sum initialized to the identity operand associated with the operator that comes into play
  - Here we have “+”, so it’s a zero (0)

All local copies of `sum` added together and stored in “global” variable
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);
}
```
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;
printf("Value of integral is: %f\n", my_pi);

return 0;
}
```
OpenMP Reduction Example:

Output

[negrut@euler24 CodeBits]$ g++ testOMP.cpp -o me759.exe
[negrut@euler24 CodeBits]$ ./me759.exe 100000
Value of integral is: 3.141593
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>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Initial Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example: 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 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);
        }
        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>Declared 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:
    ```bash
    # 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

- Q: How would you fix this issue with the static variable?
  - Not necessarily to print the values in increasing order, but to make sure there are no race conditions
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

- It 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:
    ```bash
    # 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
  - No reason for this behavior to change
Concluding Remarks on the OpenMP API
OpenMP: 30,000 Feet Perspective

- Good momentum behind OpenMP owing to the ubiquity of the multi-core chips
- Shared memory, thread-based parallelism
- Relies on the programmer defining parallel regions
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997
  - OpenMP 4.0 – complete specifications released in July 2013
OpenMP
The 30,000 Feet Perspective

- Nomenclature:
  - Multicore Communication API (MCAPI)
  - Multicore Resource-sharing API (MRAPI)
  - Multicore Task Management API (MTAPI)
The OpenMP API

- The OpenMP API is a 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"`
The OpenMP API: The Directives

[Cntd.]

- The “directives” fall 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`
OpenMP 4.0:
Subset of Run-Time Library OpenMP Routines

1. omp_set_num_threads
2. omp_get_num_threads
3. omp_get_max_threads
4. omp_get_thread_num
5. omp_get_thread_limit
6. omp_get_num_procs
7. omp_in_parallel
8. omp_set_dynamic
9. omp_get_dynamic
10. omp_set_nested
11. omp_get_nested
12. omp_set_schedule
13. omp_get_schedule
14. omp_set_max_active_levels
15. omp_get_max_active_levels
16. omp_get_level
17. omp_get_ancestor_thread_num
18. omp_get_team_size
19. omp_get_active_level
20. omp_init_lock
21. omp_destroy_lock
22. omp_set_lock
23. omp_unset_lock
24. omp_test_lock
25. omp_init_nest_lock
26. omp_destroy_nest_lock
27. omp_set_nest_lock
28. omp_unset_nest_lock
29. omp_test_nest_lock
30. omp_get_wtime
31. omp_get_wtick
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
[select env variables]

- **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`
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
OpenMP, Some Caveats

- There is a lag between the moment a new specification is released and the time a compiler is capable of handling all of its aspects
  - Intel’s compiler is probably most up to speed

- 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/](https://computing.llnl.gov/tutorials/openMP/)
- OpenMP.org, [http://openmp.org/](http://openmp.org/)
- OpenMP 3.0 API Summary Cards:
  - C/C++: [http://openmp.org/mp-documents/OpenMP-4.0-C.pdf](http://openmp.org/mp-documents/OpenMP-4.0-C.pdf)
- [http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf](http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf)
Multi-Core Computing, Next Decade
The Price of 1 MFlops

- 1 Mflops: 1 million floating point operations per second

  - 1961:
    - One would have to combine 17 million IBM-1620 computers to reach 1 Mflops
    - At $64K apiece, when adjusted for inflation this would be ½ the 2013 US national debt

  - 2000:
    - About $1,000

  - 2013:
    - Less than 20 cents out of the value of a workstation
Feature Length on a Chip: Moore’s Law at Work

- 2013 – 22 nm
- 2015 – 14 nm
- 2017 – 10 nm
- 2019 – 7 nm
- 2021 – 5 nm
- 2023 – ??? (carbon nanotubes?)
What Does This Mean?

● One of two things:

  ● We either increase the computational power of the chip since you have more transistors

  ● Alternatively, we can keep the number of transistors constant but decrease the size of the chip
Increasing the Number of Transistors: Multicore is Here to Stay

- What does that buy us?
- More computational units

October 2013:
- Intel Xeon w/ 12 cores – 3 billion transistors (today’s top of the line)
- 0.2 Tflops, give or take

Projecting ahead:
- 2015: 24 cores
- 2017: about 50 cores
- 2019: about 100 cores
- 2021: about 200 cores
Decreasing the Area of the Chip

- Decreasing the chip size: imagine that we want to pack the power of today’s 12 core chip onto tomorrow’s wafer

- Size of chip – assume a square of length “L”
  - 2013: L is about 20 mm
  - 2015: L ≈ 14 mm
  - 2017: L ≈ 10 mm
  - 2019: L ≈ 7 mm
  - 2021: L ≈ 5 mm → a fifth of an inch fits on your phone
Mechanical Engineering: GPU Computing Example
Mechanical Engineering: OpenMP Example
Mechanical Engineering:
MPI Computing Example
End: Parallel Computing w/ OpenMP
Beginning: Parallel Computing w/ MPI