“If a program manipulates a large amount of data, it does so in a small number of ways.”

Alan J. Perlis
Before We Get Started…

- Last lecture
  - Parallel tasks
  - Variable scoping: private & sharing variables
  - Synchronization issues, atomic operations, global operations

- Today
  - OpenMP example, variable scoping
  - Wrap-up, OpenMP
  - CUDA, OpenMP, MPI – putting things in perspective

- Looking ahead
  - One more regular lecture on Th: parallel programming patterns
  - Guest lectures next week, on CUDA and OpenMP
Select your time slot through doodle poll: [http://www.doodle.com/3b65mca99d6wuety](http://www.doodle.com/3b65mca99d6wuety)
- Date & time-slot selection: first-come-first-to-choose
- Time slots: 30 mins long
  - 20 mins presentation + 10 mins follow up questions
- All presentations held in 3164ME

Final Project document
- No longer than 12 pages, font no smaller than 10 pt
  - Statement of the problem solved, brief review of literature, state your approach, present results, end with conclusions and directions of future investigation
- Final Project due on Wd, May 16 at 11:59 PM
Exercise: Variable Scoping Aspects

- Consider parallelizing the following code

```c
#include <omp.h>
#include <math.h>
#include <stdio.h>

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);

    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

- Example obviously contrived but helps to understand the scope of different variables

```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)

- Conclusion: be careful when you work with static or some other globally shared variable 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…
    Still don’t understand why this is the case (it’s surprising it works like this…)
OpenMP API
OpenMP: The 30,000 Feet Perspective

OpenMP Components

- Parallel control structures
  - Governs flow of the control in the program
    - parallel
    - single

- Work Sharing
  - Distributes work among threads
    - for
    - section
    - task

- Data Environment
  - Scopes variables
    - shared
    - private

- Synchronization
  - Coordinates thread execution
    - critical
    - atomic
    - barrier

- Runtime functions, env. variables
  - Controls runtime environment
    - omp_set_num_threads()
    - omp_get_thread_num()
    - OMP_NUM_THREADS
    - OMP_SCHEDULE
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"`

[R. Harman-Baker]
The OpenMP API Directives

- Directives (or Pragmas) used to
  - Express/Define parallelism (flow control)
    - Example: `#pragma omp parallel for`
  - Specify data sharing among threads (communication)
    - Example: `#pragma omp parallel for private(x,y)`
  - Synchronization (coordination or interaction)
    - Example: `#pragma omp barrier`
## OpenMP 3.1:
Summary 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`
```c
#include <stdio.h>
#include <omp.h>

int main() {
    omp_set_num_threads(4);
    printf_s("First call: %d\n", omp_get_num_threads( ));
    #pragma omp parallel
        #pragma omp master
        {
            printf_s("Second call: %d\n", omp_get_num_threads( ));
        }
    printf_s("Third call: %d\n", omp_get_num_threads( ));

    #pragma omp parallel num_threads(3)
        #pragma omp master
        {
            printf_s("Fourth call: %d\n", omp_get_num_threads( ));
        }
    printf_s("Last call: %d\n", omp_get_num_threads( ));
    return 0;
}
```
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
[recent ones, added as of 3.0]

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

- **OMP_WAIT_POLICY**
  - Provides hint to an OpenMP implementation about 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
Concluding Remarks & Wrap-up
OpenMP Summary

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

- Industry-standard shared memory programming model
  - First version released in 1997

- OpenMP Architecture Review Board (ARB) determines updates to standard
  - The final specification of Version 3.1 released in July of 2011 (minor update)
OpenMP, Summary

- OpenMP provides small yet versatile programming model
  - This model serves as the inspiration for the OpenACC effort to standardizing approaches that can factor in the presence of a GPU accelerator

- Not at all intrusive, very straightforward to parallelize existing code
  - Good efficiency gains achieved by using parallel regions in an existing code

- Work-sharing constructs: `for`, `section`, `task` enable parallelization of computationally intensive portions of program
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

- It seems that the vendors lag behind when it comes to support of the latest OpenMP specifications
  - Intel probably is most up to speed although I haven’t used their compilers

- OpenMP threads are heavy
  - Very good for handling parallel tasks
  - Not particularly remarkable at handling fine grain data parallelism (vector architectures excel here)
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 3.1 API Summary Cards:
CUDA, OpenMP, MPI: 
Putting Things in Perspective
Pros, CUDA

- Many remarkable success stories when the application targeted is data parallel and with high arithmetic intensity
  - One order of magnitude speed-ups are common

- Very affordable – democratization of parallel computing
  - At a price of $10K you get half the flop rate of what an IBM BlueGene/L got you six or seven years ago

- Ubiquitous
  - Present on more than 100 million computers today support CUDA

- Very good productivity tools
Cons, CUDA

- To extract last ounce of performance that makes GPU computing great you need to understand the computational model and the underlying hardware.

- Not that much device memory available – 6 GB is the most you get today
  - Getting around it requires moving data in and out of the device, which complicates the programming job.

- Until the CPU and GPU are fully integrated, the PCI connection is impacting performance and complicating the implementation task.

- For true HPC, using CUDA in conjunction with MPI remains a challenge
  - Ongoing projects aimed at addressing this, but still…
What Would Be Nice…

- The global memory bandwidth should increase at least as fast as the rate at which the number of scalar processors increases.

- Integrate CPU & GPU so that concept of global device memory disappears.

- Have the OpenACC standard succeed for seamless parallel accelerator and/or many-core programming.
Pros of OpenMP

- Because it takes advantage of shared memory, the programmer does not need to worry (that much) about data placement.
- Programming model is “serial-like”, thus conceptually simpler than message passing.
- Compiler directives are generally simple and easy to use.
- Legacy serial code does not need to be rewritten.
Cons of OpenMP

- The model doesn’t scale up all that well

- In general, only moderate speedups can be achieved
  - Because OpenMP codes tend to have serial-only portions, Amdahl’s Law prohibits substantial speedups

- Amdahl’s Law:
  \[ s = \text{Fraction of serial execution time that cannot be parallelized} \]
  \[ N = \text{Number of processors} \]

  Execution speedup: \[ \frac{1}{s + \frac{1-s}{N}} \]

- If you have big loops that dominate execution time, these are ideal targets for OpenMP
Pros of MPI

- Very good vendor support for the standard
  - It was great that the community converged upon a standard (something that can’t be said about GPU computing)

- Proven parallel computing solution, demonstrated to scale up to hundreds of thousands of cores

- Can be deployed both for distributed as well as shared memory architectures

- Today it is synonym with High Performance Computing
  - Provided a clear and relatively straightforward framework for reaching Petaflops grade computing
Cons of MPI

- The interconnect is Achilles' heel. Top bandwidths today are comparable to what you get over PCI-Express
  - Latency typically worse though

- Like CUDA, works well only for applications where you don’t have to communicate all that much (high arithmetic intensity)
General Remarks on Parallel Computing

- Parallel Computing has evolved a lot since the days of posix threads

- Nonetheless, it continues to be challenging
  - Switching your thinking about getting a job done from sequential to parallel mode takes some time but it’s a skill that is eventually acquired
    - Parallel Programming more difficult than programming for Sequential Computing
  - Productivity tools (debuggers, profilers, build solutions) more challenging to master
  - Need to understand the problem that you solve, the pros/cons of the parallel programming models available, and of the hardware on which your code will run
Skills I hope You Picked Up in ME964

- I think of these as items that you can add to your resume:
  - Basic understanding of hardware for parallel computing
  - Basic understanding of parallel execution models: SIMD, MIMD, etc.
  - CUDA programming
  - OpenMP Programming
  - MPI Programming
  - Build management: CMake
  - Integrated development environment (IDE): Eclipse
  - Version Control: Mercurial, git
  - Debugging: gdb, cuda-gdb, memcheck, cuda-memcheck, valgrind
  - Profiling: nvvp
Course Evaluation

- Please take 10 minutes to indicate what you liked and what needs to be changed in ME964
  - I *very* much appreciate your feedback, you help shaping up this course

- I plan to upload your forms online on the course website to help future students make an informed decision in relation to registering for ME964