Multicore Parallel Computing with OpenMP

Dan Negrut
University of Wisconsin - Madison

Darmstadt Technical University – August 2016
Short Recap, Looking Back at Wd

- Wrap up of GPU computing
  - Thread divergence
  - Memory ecosystem
    - Registers
    - Shared memory (example of matrix multiplication)
    - Global memory accesses (coalesced accesses)
- Synchronizing execution
- Coordinating memory transactions w/ atomic operations
- Optimization issues (occupancy, rules of thumb)
- Using libraries to do GPU computing: using thrust

- Supplemental material was provided (not covered)
  - Example: Uniform memory in CUDA
Today’s Plan

- Multi-core parallel computing on one workstation/laptop
- Focus on support provided by OpenMP
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 – ???
Small Feature Length: What Does This Mean?

- One of two things:
  - You either increase the computational power and/or smarts of the chip since you have more transistors, or
  - You 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 you?

- More computational units

- October 2015:
  - Intel Xeon w/ 18 cores – 5.7 billion transistors (E7-8890 v3, $7200)
  - Intel Xeon Phi: about 60 lesser cores

- Projecting ahead (not going to happen, “dark silicon” phenomenon):
  - 2017: about 36 cores
  - 2019: about 60 cores
  - 2021: about 100 cores
Decreasing the Area of the Chip

- Decreasing the chip size: imagine that you want to pack the power of today’s 12 core chip on 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
Dennard’s Scaling: Making Moore’s Law Useful

- Dennard scaling: dictates how the voltage/current, frequency should change in response to our etching ability

- Voltage lower and lower, static power losses more prevalent

- Transistors are too close + narrower dielectric
  - Many leaky transistors them per unit area

- Amount of power dissipated grows to high levels
  - Thermal runaway
  - Computing not reliable anymore: what’s noise and what’s information?
Power Issues: The Dark Side

- Dark Silicon: transistors that manufacturers cannot afford to turn on
- Too many transistors on a chip – cannot afford to power them lest the chip melts

Illustrating Hardware

- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Sophisticated micro-architecture structure for ILP acceleration
  - Superscalar
  - Supports HTT (hyper-threading technology)

Good source of information for these slides:
http://www.realworldtech.com/
Superscaling vs. Pipelining

Baseline  1 instruction / 5 clocks
Superscaling (2 execution units)  2 instructions / 5 clocks
Pipelining  1 instruction / clock
Combined  2 instructions / clock

[Wikipedia]→
Brain Teaser

- Can you think of a code snippet that would qualify for the instruction issue shown on the previous slide?
Illustrating Hardware

- Actual layout of the chip

![Chip Layout Image]

- Schematic of the chip organization
- LLC: last level cache (L3)
- Three clocks:
  - A core’s clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
  - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
  - Ring bus and the shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores
Caches

- **Data:**
  - L1 – 32 KB per *core*
  - L2 – 512 KB or 1024 KB per *core*
  - L3 – 8 MB per *CPU*

- **Instruction:**
  - L0 – room for about 1500 microoperations (uops) per core
    - See H/S primer, online
  - L1 – 32 KB per core

- **Cache is a black hole for transistors**
  - Example: 8 MB of L3 translates into:
    - $8 \times 1024 \times 1024 \times 8 \text{ (bits)} \times 6 \text{ (transistors per bit, SRAM)} = 402 \text{ million transistors out of 1.4 billions}$

- **Caches are *very* important for good performance**
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads.
- One warp issued at each clock cycle by each scheduler.
- During no cycle can more than 2 warps be dispatched for execution on the four functional units.
- Scoreboarding is used to figure out which warp is ready.
Haswell Microarchitecture
[30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Turning CISC to RISC
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blue)
    - Memory related (green)

- More details - see primer posted online:
  [http://www.realworldtech.com](http://www.realworldtech.com)
  [http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf](http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf)
Haswell Microarchitecture: The HTT Feature

- Employs the so called Hyper-Threading Technology (HTT) on each core of the chip

- Relatively similar to what NVIDIA does with the SM
  - The SM is overcommitted – there are up to 64 warps waiting to execute to hide latencies

- HTT: two threads active at any given time and the scheduler tries to issue instructions from both
  - The HW upgrades to support HTT are relatively minor
    - Required to save execution state of a thread when it’s idle and not running for lack of data of lack of functional units available
HTT, Nuts and Bolts

- A processor core to maintain two architectural states, each of which can support its own thread.

- Many of the internal microarchitectural hardware resources are shared between the two threads.
HTT

- Essentially, one physical core shows up as two virtual cores

- Why is it good?
  - More functional units are used at any given time
  - Instruction dependencies are reduced since instructions executed belong to different threads

- When one thread has a cache miss, branch mispredict, or any other pipeline stall, the other thread continues processing instructions at nearly the same rate as a single thread running on the core
Moving from HW to SW
Acknowledgements

- Many OpenMP slides are from Intel’s library of presentations for promoting OpenMP
  - Slides used herein with permission
  - Credit given where due: IOMPP
    - IOMPP stands for “Intel OpenMP Presentation”

- However, there are several other sources
  - Source listed in the left lower corner
  - Tried to give credit where due, apologies if I overlooked somebody
Shared Memory Multi-Processing [SMMP]

- Threads have access to large pool of shared memory
- Threads can have private data
  - Not accessible by other threads
- Data transfer/access transparent to programmer
- Synchronization is implicit but can be made explicit as well

[CodeProject]→
SMMP Architecture Flavors: UMA & NUMA

- UMA: “uniform memory access” architecture
  - Each processor has “identical time access” to the memory
  - AKA SMP: “Symmetric” Multi-Processor
  - SMP a particular case of SMMP

- NUMA: “non-uniform memory access” architecture
  - A processor has faster access to a portion of the memory compared to the rest of the processors
Brain Teaser

- Can you think of an example of an UMA architecture?

- Can you think of an example of an NUMA architecture?
Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element needs to be processed to produce a result.
  - When this processing can be done in parallel, we have data parallelism.
  - Example:
    - Filtering a picture; i.e., processing each pixel out of a million in a snapshot.

- **Task parallelism**
  - You have a collection of tasks that need to be completed.
  - If these tasks can be performed in parallel you are faced with a task parallel job.
  - Examples:
    - Microwave a soup, make a salad, boil pasta, bake a cake.
    - All of the above can happen at the same time.
OpenMP
MULTICORE PARALLEL COMPUTING ON THE CPU
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization
- Advanced topics
Work Plan

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

- **Advanced topics**
OpenMP: Target Hardware

- CUDA: targeted parallelism on the GPU

- OpenMP: targets parallelism on shared memory multiprocessor architectures
  - Handy when
    - You have a machine that has 20 cores
    - You have a large amount of shared memory, say 128GB

- MPI: targets parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
What Is OpenMP?

- Portable, shared-memory threading API
  - Bindings: Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows

- Standardizes task & loop-level parallelism
- Very good at coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 25 years of compiler-directed threading experience

- Current spec is OpenMP 4.0
  - Released in 2013
  - [http://www.openmp.org](http://www.openmp.org)
  - More than 300 Pages
History, OpenMP

- 1997: OpenMP 1.0 for FORTRAN
- 1998: OpenMP 1.0 for C and C++
- 1999: OpenMP 1.1 for FORTRAN (errata)

- 2000: OpenMP 2.0 for FORTRAN
- 2002: OpenMP 2.0 for C and C++
- 2005: OpenMP 2.5 includes both programming languages.

- 05/2008: OpenMP 3.0 release
- 07/2011: OpenMP 3.1 release
- 07/2013: OpenMP 4.0 release

[Dirk Schmidl, Christian Terboven]→
OpenMP: What’s Reasonable to Expect

- If you have more than 15 cores or so, it’s pretty unlikely that you can get a speed-up on that scale. All sorts of overheads kick in to slow you down
  - Beyond 15: law of diminishing return

- Some reasons: no overcommitment of HW, false cache sharing, cache coherence, etc.

- A word on lack of overcommitment
  - Recall the trick that helped the GPU hide latency
    - Overcommitting an SM and hoping to hide memory access latency with warp execution
  - This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what’s offered by HTT
OpenMP Programming Model

- Master thread spawns a team of threads as needed
  - Managed transparently on your behalf
  - It relies on low-level thread fork/join methodology to implement parallelism
  - The developer is spared the details

- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program
OpenMP: Directives-based API

- Most OpenMP constructs are compiler directives or pragmas
  - For C and C++, the pragmas take the form:
    `#pragma omp construct [clause [clause]...]`
  - For Fortran, the directives take one of the forms:
    `C$OMP construct [clause [clause]...]`
    `!$OMP construct [clause [clause]...]`
    `*$OMP construct [clause [clause]...]`
OpenMP: Library Support

- Access to the API - header file in C, or Fortran 90 module
  ```
  #include "omp.h"
  use omp_lib
  ```

- Runtime 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()
    ```
  - Many more...

[IOOMP]→

https://computing.llnl.gov/tutorials/openMP/
OpenMP: Environment variables

- Controlling the number of threads
  The default number of threads that a program uses when it runs is the number of processors on the machine
  - For the C shell: `setenv OMP_NUM_THREADS number`
  - For the Bash shell: `export OMP_NUM_THREADS=number`
  - For the Windows shell: `set OMP_NUM_THREADS=number`

- Schedule for parallel for loops (schedule(runtime))
  - `export OMP_SCHEDULE="static,1"`

- Many more, visiting later…
Why Compiler Directive and/or Pragmas?

- One of OpenMP’s design principles: the same code, with no modifications, can run either on an one core machine or a multiple core machine.
- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas.
- These directives picked up by the compiler only if you instruct it to compile in OpenMP mode:
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code.
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>
Compiling Using the Command Line

Method depends on compiler

- **GCC:**
  
  ```
  $ g++ -o integrate_omp integrate_omp.c -fopenmp
  ```

- **ICC:**
  
  ```
  $ icc -o integrate_omp integrate_omp.c -openmp
  ```

- **MSVC (not in the express edition):**
  
  ```
  $ cl /openmp integrate_omp.c
  ```
OpenMP: Timing

- Timing:

```c
#include <omp.h>
stime = omp_get_wtime();
mylongfunction();
etime = omp_get_wtime();
total = etime - stime;
```
Work Plan

- What is OpenMP?
  - Parallel regions
    - Work sharing
    - Data environment
    - Synchronization
- Advanced topics
Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to structured blocks
  - **structured block**, definition: a block with one point of entry at the top and one point of exit at the bottom
  - The only “branches” allowed are exit() function calls

A structured block

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job (id);
    if ( not_conv(res[id]) ) goto more;
}
printf ("All done\n");
```

Not a structured block

```c
if (go_now()) goto more;
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if ( conv (res[id]) ) goto done;
    goto more;
}
done: if (!really_done()) goto more;
```

There is an implicit barrier at the right “}” curly brace and that’s the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int myId = omp_get_thread_num();
        int nThreads = omp_get_num_threads();

        printf("Hello World. I'm thread %d out of %d.\n", myId, nThreads);
        for( int i=0; i<2 ;i++ )
            printf("Iter:%d\n",i);
    }
    printf("All done here...\n");
}

Example: Hello World

Here’s my laptop:
Intel Core i5-3210M @ 2.50GHz 3 MB L3 Cache, TDP 35 Watts, Two-Core Four-Thread Processors
OpenMP: Important Remark

- One of the key tenets of OpenMP is that of data independence across parallel jobs.

- Specifically, when distributing work among parallel threads it is assumed that there is no data dependency.

- Since you place the `omp parallel` directive around some code, it is your responsibility to make sure that data dependency is ruled out.
  - Compilers are not smart enough and sometimes they cannot identify data dependency between what might look like independent parallel jobs.
Work Plan

- What is OpenMP?
  - Parallel regions
    - Work sharing
  - Data environment
  - Synchronization
- Advanced topics
Work Sharing

- **Work sharing** is the general term used in OpenMP to describe distribution of work across threads.

- Three primary avenues for work sharing in OpenMP:
  - `omp for` construct
  - `omp sections` construct
  - `omp task` construct

Each of them *automatically* divides work among threads.
Work Plan

- What is OpenMP?
  Parallel regions
  **Work sharing – Parallel for loops**
  Data environment
  Synchronization

- Advanced topics
The **omp for** Directive

// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1; i <= N; i++)
        c[i] = a[i] + b[i];

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct

[example above assumes three threads are in the thread team]
Combining Constructs

- These two code segments are equivalent

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i< MAX; i++) {
        res[i] = someFunc(i);
    }
}

#pragma omp parallel for
for (int i=0; i< MAX; i++) {
    res[i] = someFunc(i);
}
```

There is an implicit barrier here.
OpenMP: Important Remark

- Key tenets of OpenMP: that of *data independence* in parallel regions
  - Danger: race conditions

- Therefore, when distributing work among parallel threads it is assumed that there is no data dependency

- Since you place the `omp parallel for` directive around some code, it is your responsibility to make sure that data dependency is ruled out
  - Compilers many times can’t identify data dependency between what might look as independent parallel jobs
The private Clause

- Reproduces the variable for each task
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
  - The value that Thread_1 stores in \( x \) is different than value that Thread_2 stores in variable \( x \)
  - Variables are un-initialized; C++ object is default constructed

```c
void* work(float* c, int N) {
    float x, y;
    int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```
Default Partitioning: Possible Problems

- Most OpenMP implementations use as default block partitioning
  - Each thread is assigned roughly $n/\text{thread\_count}$ iterations

- This may lead to load imbalance when the work per iteration varies

```c
sum = 0;
for(i = 0; i <= n; i++)
    sum += f(i);
```

(assume the time required by a call to $f(i)$ is proportional to $i$)
The schedule Clause

- The `schedule` clause affects how loop iterations are mapped onto threads

**schedule(static [,chunk])**
- Blocks of iterations of size “chunk” assigned to each thread
- Round robin distribution
- Low overhead, may cause load imbalance

**schedule(dynamic[,chunk])**
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

**schedule(guided[,chunk])**
- Dynamic schedule starting with large block
- Size of the blocks shrink rather fast; no smaller than “chunk”
**schedule Clause Example**

```c
#pragma omp parallel for schedule(static, 8)
    for(int i = start; i <= end; i += 2)
    {
        if (TestForPrime(i)) gPrimesFound++;
    }
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

```
i={3,5,7,9,11,13,15,17}
```
Loop scheduling

iteration

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
STATIC  STATIC, 3  DYNAMIC [,1]  DYNAMIC, 3  GUIDED [,1]

T0  T1  T2
Choosing a Schedule

- **STATIC** is best for balanced loops – least overhead.

- **STATIC**, \( n \) good for loops with mild or smooth load imbalance
  - Prone to introduce “false sharing” (discussed later)

- **DYNAMIC** useful if iterations have widely varying loads
  - Prone to adversely impact data locality (cache misses)
  - That’s because threads are assigned to available cores, NUMA comes into play

- **GUIDED** often less expensive than **DYNAMIC**
  - Beware of loops where first iterations are the most expensive
What for loops can be parallelized?

- OpenMP will only parallelize for loops that are in **canonical form**

\[
\begin{aligned}
\text{for} & \quad \text{index} = \text{start} \ ; \\
& \quad \text{index} < \text{end} \\
& \quad \text{index} \leq \text{end} \\
& \quad \text{index} \geq \text{end} \\
& \quad \text{index} > \text{end} \\
& \quad \text{index} + + & \quad \text{index} + + \\
& \quad \text{index} -- & \quad \text{index} --
\end{aligned}
\]
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing – Parallel sections
  Data environment
  Synchronization

● Advanced topics
Function Level Parallelism

```c
a = alice();
b = bob();
s = boss(a, b);
k = kate();
printf("%6.2f\n", bigboss(s,k));
```

alice, bob, and kate can be computed in parallel
Parallel `omp sections` Directive

- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing $N$ sub-blocks of code that may be executed concurrently by $N$ threads
- Encompasses all `omp section` blocks, see below

- `#pragma omp section`
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads

There is an “s” here

There is no “s” here
Functional Level Parallelism Using `omp sections`

```
#pragma omp parallel sections
{
#pragma omp section
  a = alice();
#pragma omp section
  b = bob();
#pragma omp section
  k = kate();
}

double s = boss(a, b);
printf("%6.2f\n", bigboss(s,k));
```

There is an implicit barrier here.

Is there another way to parallelize this?
Advantage of Parallel Sections

- Independent sections of code can execute concurrently → reduces execution time

```c
#pragma omp parallel sections
{
#pragma omp section
    phase1();
#pragma omp section
    phase2();
#pragma omp section
    phase3();
}
```

The pink and green tasks are executed at no additional time-penalty in the shadow of the blue task.
#include <stdio.h>
#include <omp.h>

void spin_in_place(double duration) {
    double start_time = omp_get_wtime();
    while(omp_get_wtime() - start_time < duration) {}
}

int main() {
    printf("\nUsing 2 threads on 3 sections\n");
    double start_time = omp_get_wtime();

#pragma omp parallel sections num_threads(2)
{
#pragma omp section
    { printf("Start work 1\n"); spin_in_place(2); printf("End work 1\n"); }
#pragma omp section
    { printf("Start work 2\n"); spin_in_place(2); printf("End work 2\n"); }
#pragma omp section
    { printf("Start work 3\n"); spin_in_place(2); printf("End work 3\n"); }
}

    printf("Wall clock time: %.2g\n", omp_get_wtime() - start_time);
}

return 0;
sections Example: 2 threads

```
serban@lagrange:~/CODES/OMP
[serban@lagrange:~/CODES/OMP]$ g++ -fopenmp sections_example_1.cpp
[serban@lagrange:~/CODES/OMP]$ ./a.out

Using 2 threads on 3 sections
Start work 1
Start work 2
End work 1
Start work 3
End work 2
End work 3
Wall clock time: 4
[serban@lagrange:~/CODES/OMP]$ 
```
```c
#include <stdio.h>
#include <omp.h>

void spin_in_place(double duration) {
    double start_time = omp_get_wtime();
    while(omp_get_wtime() - start_time < duration) {}
}

int main() {
    printf("\nUsing 4 threads on 3 sections\n");
    double start_time = omp_get_wtime();

    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Start work 1\n"); spin_in_place(2); printf("End work 1\n");
        }
        #pragma omp section
        {
            printf("Start work 2\n"); spin_in_place(6); printf("End work 2\n");
        }
        #pragma omp section
        {
            printf("Start work 3\n"); spin_in_place(2); printf("End work 3\n");
        }
    }

    printf("Wall clock time: %.2g\n", omp_get_wtime() - start_time);
}

return 0;
```
sections  Example: 4 threads

```
[serban@lagrange:~/CODES/OMP$  g++ -fopenmp sections_example_1.cpp
[serban@lagrange:~/CODES/OMP$  ./a.out

Using 4 threads on 3 sections
Start work 1
Start work 3
Start work 2
End work 1
End work 3
End work 2
Wall clock time: 6
[serban@lagrange:~/CODES/OMP$  
```
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing – Parallel tasks
  Data environment
  Synchronization

● Advanced topics
OpenMP Tasks

- **Task** – Most important feature added as of OpenMP 3.0 version

- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer

- Start next with a few motivating examples
Example, Static Scheduling

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

int main() {
    #pragma omp parallel for schedule (static)
    for (int i = 0; i <= 14; i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```
```c
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
    if (i <= N)
        return N;
    else
        return 0;
}

int main(){
    int upperB = 14;

    for (int i = 0; i <= getUpperBound(i,upperB); i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
```

Code run on one thread, **sequential** execution, no OpenMP
#include <stdio.h>
#include <omp.h>

int getUpperBound(int i, int N){
    if (i <= N)
        return N;
    else
        return 0;
}

int main() {
    int upperB = 14;

    #pragma omp parallel for schedule (static)
    for (int i = 0; i <= getUpperBound(i, upperB); i++){
        printf("I'm thread %d working on iteration %d\n", omp_get_thread_num(), i);
    }
    printf("All done here...\n");
}
Tasks: What Are They?

- Tasks are independent units of work.
- A thread is assigned to perform a task.
- Tasks might be executed immediately or might be deferred.
  - The OS & runtime decide which of the above.
- Tasks are composed of:
  - **code** to execute
  - **data** environment
  - **internal control variables** (ICV)
Tasks: What Are They?
[More specifics…]

- Code to execute
  - The literal code in your program enclosed by the task directive

- Data environment
  - The shared & private data manipulated by the task

- Internal control variables
  - Thread scheduling and environment variables

- More formal definition: A task is a specific instance of executable code and its data environment, generated when a thread encounters a task construct

- Two activities: (1) packaging, and (2) execution
  - A thread packages new instances of a task (code and data)
  - Some thread in the team executes the task at some later time
using namespace std;
typedef list<double> LISTDBL;

void doSomething(LISTDBL::iterator& itrtr)
{
    *itrtr *= 2.0;
}

int main()
{
    LISTDBL test; // default constructor
    LISTDBL::iterator it;

    for( int i=0;i<4;++i)
        for( int j=0;j<8;++j) test.insert(test.end(), pow(10.0,i+1)+j);
    for( it = test.begin(); it!= test.end(); it++)
        cout << *it << endl;

    it = test.begin();
    #pragma omp parallel num_threads(8)
    {
        #pragma omp single // one single thread is packaging the tasks, one at a time
        {
            while( it != test.end() ) {
                #pragma omp task private(it) // "it" private to guard against foul play by "it++"
                {
                    doSomething(it);
                }
                it++;
            }
        }
        for( it = test.begin(); it != test.end(); it++)
            cout << *it << endl;
    }
    return 0;
}
Compile like:

```bash
$ g++ -o testOMP.exe testOMP.cpp
```

Initial values...

Final values...
Task Construct – Explicit Task View

- A team of threads is created at the `omp parallel` construct.
- A single thread is chosen to execute the while loop – let’s call this thread “L”.
- Thread L runs the while loop, creates tasks, and fetches next pointers.
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it.
- Each task run by one thread.
- All tasks complete at the barrier at the end of the parallel region’s construct.
- Each task has its own stack space that will be destroyed when the task is completed.

```c
#pragma omp parallel
    // threads are ready to go now
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p!=listEnd) {
            // block 2
            #pragma omp task private(p)
            process(p);
            p = p->next;  // block 3
        }
    }
}
```
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
//threads are ready to go now
{
    #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
        }
    }
}
```

How about synchronization issues?
Tasks: Synchronization Issues

- **Setup:**
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guarantee 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
**Comments: section vs. task**

- **sections** have a "static” attribute
  - Things are mostly settled at compile time

- The **task** construct is more recent and more sophisticated
  - They have a “dynamic” attribute: things are figured out at run time and the construct counts under the hood on the presence of a scheduling agent
  - They can encapsulate any block of code
    - Can handle nested loops and scenarios when the number of jobs is not clear
  - The runtime generates and executes the tasks, either at implicit synchronization points in the program or under explicit control of the programmer

- **NOTE:** It’s the developer responsibility to ensure that different tasks can be executed concurrently; i.e., there is no data dependency
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing – Parallel tasks
    - Data environment
    - 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
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
  - Data is initialized using the value of the variable of same name from the master thread
Example:
private vs. firstprivate

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

int main(void) {
    int i = 10;

    #pragma omp parallel private(i)
    {
        int threadID = omp_get_thread_num();
        printf("thread %d: i = %d\n", threadID, i);
        i = 1000 + threadID;
    }

    printf("i = %d\n", i);

    return 0;
}
```
Example:
private vs. firstprivate

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

int main(void) {
    int i = 10;

#pragma omp parallel firstprivate(i)
    {
        int threadID = omp_get_thread_num();
        printf("threadID + i = %d\n", threadID+i);
    }

    printf("i = %d\n", i);

    return 0;
}
```

[stackoverflow]→
There is a lastprivate flavor of private variable

The enclosing context's version of the variable is set equal to the private version of whichever thread executes the final iteration of the work-sharing construct (for, section, task)
Data Scoping – The Basic Rule

- When in doubt, explicitly indicate who’s what

[Data scoping: one of the most common sources of errors in OpenMP]
```c
#pragma omp parallel shared(a,b,c,d,nthreads) private(i,tid)
{
    tid = omp_get_thread_num();
    if (tid == 0) {
        nthreads = omp_get_num_threads();
        printf("Number of threads = %d\n", nthreads);
    }

    printf("Thread %d starting...\n", tid);

#pragma omp sections nowait
{
#pragma omp section
{
    printf("Thread %d doing section 1\n", tid);
    for (i=0; i<N; i++)
    {
        c[i] = a[i] + b[i];
        printf("Thread %d: c[%d] = %f\n", tid,i,c[i]);
    }
}

#pragma omp section
{
    printf("Thread %d doing section 2\n", tid);
    for (i=0; i<N; i++)
    {
        d[i] = a[i] * b[i];
        printf("Thread %d: d[%d] = %f\n", tid,i,d[i]);
    }
} /* end of sections */

    printf("Thread %d done.\n", tid);
} /* end of parallel section */
```

Q: Do you see any problem with this piece of code?

When in doubt, explicitly indicate who’s what
A Data Environment Exercise

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 int count;
    <...>
}

A, index, count are shared by all threads, but temp is local to each thread
# Tasks and Data Scoping Issues

## Parallelization of fib()

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

int fib(int);

int main()
{
    int n = 10;
    omp_set_num_threads(4);

    #pragma omp parallel
    {
        #pragma omp single
        printf("fib(%d) = %d\n", n, fib(n));
    }
}
```

\[
F_0 = 0 \\
F_1 = 1 \\
F_n = F_{n-1} + F_{n-2}, n \geq 2
\]
Example: fib()

Assume that the parallel region exists outside of fib and that fib and the tasks inside it are in the dynamic extent of a parallel region.

```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;
}
```

What's wrong here?

- `n` is private in both tasks
- `x` is a private variable
- `y` is a private variable
- This is very important here
- What's wrong here?
Example: fib()

```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
}
```

Values of the private variables are not available outside of tasks

x is a private variable
y is a private variable

What's wrong here?
Example: `fib()`

```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` and `y` are now shared.
- We need both values to compute the sum.

The values of the `x` & `y` variables will now be available outside each task construct – after the taskwait.
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing – Parallel tasks
  - 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* directive
    - *single* directive
The **nowait** Clause

- Use when threads unnecessarily wait between independent computations

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

```c
#pragma omp single nowait
{ [...] }
```

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

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

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

```c
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A, B); // input is A, output is B

#pragma omp barrier

    DoSomeWork(B, C); // input is B, output is C
}
```
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 here?
Race Condition

- Definition, race condition: two or more threads access a shared variable at the same time.
  - Leads to nondeterministic behavior

- For example, suppose that area is shared and both Thread A and Thread B are executing the statement

```c
area += 4.0 / (1.0 + x*x);
```
Two Possible Scenarios

Value of area

Thread A
11.667
+3.765
15.432
15.432
18.995

Thread B

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

Credit: IOMPP
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;
}
```
The critical Directive

#pragma omp critical[(name)]
defines a critical region on a structured block

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

Naming the critical construct RES_lock is optional but highly recommended.

```c
float RES = 0;
#pragma omp parallel
{
#pragma omp for
    for(int i=0; i<niters; i++) {
        float B = job1(i);

#pragma omp critical(RES_lock)
        consum(B, RES);

        job2(B);
    }
}
```
The atomic Directive

- Applies only to simple update of memory location
- Special case of a critical section
  - Atomic introduces less overhead than critical

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

- Unlike a critical directive, the atomic directive
  - Can only protect a single assignment
  - Applies only to simple update of memory location

- The assignment that can be protected by atomic must be one of:
  - `x <op>= <expression>;`
  - `x++;`
  - `++x;`
  - `x--;`
  - `--x;`

- `<op>` can be one of: `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, `>>`
- `<expression>` must not reference `x`
- Only the load and store of `x` is protected (not `<expression>`)
Synchronisation

- Barriers can be very expensive
  - Typically 1000s cycles to synchronise

- Avoid barriers via:
  - **Careful** use of the `nowait` clause
  - Parallelise at the outermost level possible
    - May require re-ordering of loops +/- indexes
  - Choice of `critical` / `atomic` / lock routines may impact performance

Credit: Alan Real
Reduction Example

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

- 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
The reduction Clause

#pragma omp for 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
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>
Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4}{1 + x^2} \, dx = \pi \]

\begin{verbatim}
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);
}
\end{verbatim}
Reduction Example: Numerical Integration

```c
#include <cstdio>
#include <cstdlib>
#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 test.exe
[negrut@euler24 CodeBits]$ ./test.exe 100000
Value of integral is: 3.141593
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing – Parallel tasks
  Data environment
  Synchronization

● Advanced topics
OpenMP – Performance Issues
Performance

- Easy to write OpenMP yet hard to write an efficient program

- Five main causes of poor performance:
  - Sequential execution domination
  - Ills of shared memory and cache coherence
  - Data affinity
  - Load imbalance
  - Synchronisation
  - Compiler (non-)optimisation.
Sequential Execution Domination

- Consequence of Amdahl’s law: Parallel code will never run faster than the parts which can only be executed in serial.

- Go back and understand whether you can approach the solution from a different perspective that exposes more parallelism.

- Thinking within the context of OpenMP
  - All code outside of parallel regions and inside `master`, `single` and `critical` directives is sequential.
  - This code should be as small as possible.

Credit: Alan Real
ills of shared memory and cache coherence

- On shared memory machines, which is where OpenMP operates, communication is “disguised” as increased memory access costs.
  - It takes longer to access data in main memory or another processor’s cache than it does from local cache.

- Memory accesses are expensive
  - ~100 cycles for a main memory access compared to 1-3 cycles for a flop.

- Unlike message passing, communication is spread throughout the program
  - Hard to analyse and monitor

Credit: Alan Real
Caches and Coherency

- Shared memory programming assumes that a shared variable has a unique value at a given time

- Speeding up sequential computation: done through use of large caches

- Caching consequence: multiple copies of a physical [main] memory location may exist at different [cache] hardware locations

- For program correctness, caches must be kept coherent

- Coherency operations are usually performed on the cache lines in the level of cache closest to memory
  - LLC last level cache: high end systems these days: LLC is level 3 cache
    - Can have 45 MB of L3 cache on a high end Intel CPU

- There is much overhead that goes into cache coherence
Memory hierarchy

Credit: Alan Real
What Does MESI Mean to You?
**MESI: Invalidation-Based Coherence Protocol**

- Cache lines have **state** bits.
- Data migrates between processor caches, state transitions maintain coherence.

**MESI** Protocol has four states: **M**: Modified, **E**: Exclusive, **S**: Shared, **I**: Invalid

1. A reads “p”
   - Processor A’s Cache: I \(\rightarrow\) E
     - “exclusive”
   - Processor B’s Cache: I \(\rightarrow\) S
     - “shared”

2. B reads “p”
   - Processor B’s Cache: I \(\rightarrow\) I
     - “shared”

3. A writes “p”
   - Processor A’s Cache: S \(\rightarrow\) M
     - “modified/ dirty”
   - Processor B’s Cache: S \(\rightarrow\) I
     - “invalid”

Cache line was invalidated
Coherency – Simplified Further
[cooked up example]

- Further simplify MESI for sake of simple discussion on next slide
  - Assume now that each cache line can exist in one of 3 states:
    - **Exclusive**: the only valid copy in any cache
    - **Read-only**: a valid copy but other caches may contain it
    - **Invalid**: out of date and cannot be used

- In this simplified coherency model
  - A READ on an *invalid* or *absent* cache line will be cached as *read-only* or *exclusive*
  - A WRITE on a line not in an *exclusive* state will cause all other copies to be marked *invalid* and the written line to be marked *exclusive*
Coherency example

Credit: Alan Real
False sharing

- Cache lines consist of several words of data
  - For instance, one cache line can store 8 double precision values

- What happens when two processors are both writing to different words on the same cache line?
  - Each write will invalidate the other processors copy
  - Lots of remote memory accesses

- Symptoms:
  - Poor speedup
  - High, non-deterministic numbers of cache misses
  - Mild, non-deterministic, unexpected load imbalance

Credit: Alan Real
False Sharing Example

double sum = 0.0, sum_local[NUM_THREADS];
#pragma omp parallel num_threads(NUM_THREADS)
{
    int me = omp_get_thread_num();
    sum_local[me] = 0.0;

#pragma omp for
    for (i = 0; i < N; i++)
        sum_local[me] += x[i] * y[i];

#pragma omp atomic
    sum += sum_local[me];
}
Sometimes This Fixes It

- Reduce the frequency of false sharing by using thread-local copies of data.
  - The thread-local copy read and modified frequently
  - When complete, copy the result back to the data structure.

```c
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; //Frequent read/write access variable
    unsigned long start;
    unsigned long end;
};

void threadFunc(void *parameter)
{
    ThreadParams* p = (ThreadParams*)parameter;
    // local copy for read/write access variable
    unsigned long local_v = p->v;

    for (unsigned long local_dummy = p->start; local_dummy < p->end; local_dummy++)
    {
        // Functional computation, read/write the “v” member.
        // Keep reading/writing local_v instead
    }

    p->v = local_v; // Update shared data structure only once
}
```
Another Way to Fix This
[Ugly + Architecture Dependent]

- When using an array of data structures, pad the structure to the end of a cache line to ensure that the array elements begin on a cache line boundary.
- If you cannot ensure that the array is aligned on a cache line boundary, pad the data structure to twice the size of a cache line.

```c
struct ThreadParams
{
    // For the following 4 variables: 4*4 = 16 bytes
    unsigned long thread_id;
    unsigned long v; // Frequent read/write access variable
    unsigned long start;
    unsigned long end;

    // expand to 64 bytes to avoid false-sharing
    // (4 unsigned long variables + 12 padding)*4 = 64
    int padding[12];
};
__declspec(align(64)) struct ThreadParams Array[10];
```

[IOMPP]→
Data Affinity

- Data is cached on the processors which access it
  - Must reuse cached data as much as possible.

- Write code with good *data affinity*:
  - Ensure the same thread accesses the same subset of program data as much as possible

- Try to make these subsets large, contiguous chunks of data
  - Will avoid false sharing and other problems
Load Imbalance

- Load imbalance can arise from both communication and computation
- Worth experimenting with different scheduling options
  - Can use `schedule(runtime)` and the environment variable `OMP_SCHEDULE`.
- If none are appropriate, may be best to do your own scheduling

Credit: Alan Real
Synchronisation

- Barriers can be very expensive
  - Typically 1000s cycles to synchronise

- Avoid barriers via:
  - **Careful** use of the `nowait` clause
  - Parallelise at the outermost level possible
    - May require re-ordering of loops +/- indexes
  - Choice of `critical` / `atomic` / lock routines may impact performance
  - Consider using point-to-point synchronisation for nearest-neighbour type problems

Credit: Alan Real
Compiler (non)-optimization

- Sometimes the addition of parallel directives can prevent the compiler from performing sequential optimization

- Symptoms:
  - Parallel code running with 1 thread has longer execution and higher instruction count than sequential code

- Sometimes, it helps to make shared data private and local to a function

Credit: Alan Real
OpenMP – Parallelization Examples
Parallelizing a real code

- MD code
  - Original F90 code by Bill Magro
    http://www.openmp.org/samples/md.html
  - C++ version by John Burkardt
    https://people.sc.fsu.edu/~jburkardt/cpp_src/openmp/openmp.html
  - Modify code to sequential (starting point)
1. Run sequential code

Intel Core i7-6600U 2-core HHT
Windows 10
Visual Studio Ultimate 2013 (v. 12.0.40629.00 Update 5)

Baseline

\[ T_S = 47.331 \text{ s} \]
2. Profiling

![Profiler screenshot showing CPU utilization and function calls]

- **Single processor**
- **Hot spot**

The image shows a profiler output from Microsoft Visual Studio, highlighting CPU utilization and function calls. The screenshot includes a table detailing function names, total CPU usage, self-CPU usage, and total CPU time.
Hot spot
3. Parallelize loop

Add OpenMP directives to the \textbf{outer} loop over particles

\begin{verbatim}
#pragma omp parallel shared(f, nd, np, pos, vel) \ 
    private(i, j, k, rij, d, d2)
#pragma omp for reduction(+ : pe, ke)
    for (k = 0; k < np; k++) {
        // Compute the potential energy and forces.
        for (i = 0; i < nd; i++)
            f[i + k * nd] = 0.0;

        for (j = 0; j < np; j++) {
            if (k != j) {
                d = dist(nd, pos + k * nd, pos + j * nd, rij);
            }
        }
    }
\end{verbatim}
4. Run parallel code

Using 1 thread

\[
\begin{align*}
T_1 &= 55.105 \text{ s} \\
S(1) &= T_S/T_1 = 0.859
\end{align*}
\]

Reflects OpenMP overhead
Using 2 threads

$$T_2 = 31.015 \text{ s}$$

$$S(2) = T_S/T_2 = 1.526$$

$$S'(2) = T_1/T_2 = 1.669$$
Using 4 threads

\[ T_4 = 19.947 \, s \]
\[ S(4) = \frac{T_S}{T_4} = 2.373 \]
\[ S'(4) = \frac{T_1}{T_4} = 2.763 \]
5. Analyze

- Relatively large overhead: $S(1) = 0.859$ (MSVC OpenMP implementation?)

- Scaling degrades: $S(4)/4 < S(2)/2$ (HHT?)

- Amdahl’s law

$$S(N) = \frac{\tau_S + \tau_P}{\tau_S + \tau_P/N} = \frac{1}{f + (1-f)/N}$$

We find $f = 20\% - 30\%$
Parallelizing a real code

- Poisson code
  - Original C++ version by John Burkardt
    [https://people.sc.fsu.edu/~jburkardt/cpp_src/openmp/openmp.html](https://people.sc.fsu.edu/~jburkardt/cpp_src/openmp/openmp.html)
  - Modify code to sequential (as our starting point)

- Poisson equation on $[0,1] \times [0,1]$:
  \[ -\Delta U = F(x, y) \]
  \[ F(x, y) = \pi^2 (x^2 + y^2) \sin(\pi xy) \]
  with homogeneous Dirichlet BC

- Exact solution:
  \[ U = \sin(\pi xy) \]
1. Run sequential code

4 x AMD Opteron 6274 16-core
CentOS Linux release 7.1.1503
GCC 5.2.0

Baseline

\[ T_S = 173.046 \, s \]
2. Profiling

gprof

Each sample counts as 0.01 seconds.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self</th>
<th>total</th>
<th>time</th>
<th>seconds</th>
<th>seconds</th>
<th>calls</th>
<th>Ts/call</th>
<th>Ts/call</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.39</td>
<td>171.21</td>
<td>171.21</td>
<td>0.10</td>
<td>171.38</td>
<td>0.17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>171.38</td>
<td>0.00</td>
<td>128881</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>0.00</td>
<td>171.38</td>
<td>0.00</td>
<td>1</td>
<td>0.00</td>
<td>0.00</td>
<td>0.00</td>
<td>_GLOBAL__sub_I_main</td>
<td></td>
</tr>
</tbody>
</table>

Hot spot
gcov
gcov output

Hot spots
3. Parallelize loops

Add OpenMP directives to the **update** loops

```
void sweep ( int nx, int ny, double dx, double dy, double f[NX][NY],
            int itold, int itnew, double u[NX][NY], double unew[NX][NY] )
{
    int i;
    int it;
    int j;

    # pragma omp parallel shared ( dx, dy, f, itnew, itold, nx, ny, u, unew ) private ( i, it, j )
    
    for ( it = itold + 1; it <= itnew; it++ ) {
        // Save the current estimate.
        # pragma omp for
        for ( j = 0; j < ny; j++ ) {
            for ( i = 0; i < nx; i++ ) {
                u[i][j] = unew[i][j];
            }
        }
        // Compute a new estimate.
        # pragma omp for
        for ( j = 0; j < ny; j++ ) {
            for ( i = 0; i < nx; i++ ) {
                if ( i == 0 || j == 0 || i == nx - 1 || j == ny - 1 )
                    unew[i][j] = f[i][j];
                else
                    unew[i][j] = 0.25 * ( u[i-1][j] + u[i][j+1] + u[i][j-1] + u[i+1][j] + f[i][j] * dx * dy );
            }
        }
    }

    return;
}
```

**Question:** why not parallelize this outer loop?
4. Run parallel code

Using 1 thread

\[ T_1 = 174.593 \text{ s} \]
\[ S(1) = T_s/T_1 = 0.991 \]

Reflects OpenMP overhead
Using 2 threads

\[
T_2 = 86.424 \, s \\
S(2) = T_S / T_2 = 2.002 \\
S'(2) = T_1 / T_2 = 2.020
\]
Using 4 threads

\[ T_4 = 48.377 \text{ s} \]
\[ S(4) = \frac{T_S}{T_4} = 3.577 \]
\[ S'(4) = \frac{T_1}{T_4} = 3.609 \]
5. Scalability

\[ T_S = 173.046 \, s \]
OpenMP – Concluding Remarks
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 “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 API: Library Routines
(Subset of OpenMP 4.0 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

- **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/
- OpenMP.org, http://openmp.org/
- OpenMP 3.0 API Summary Cards:
  - C/C++: http://openmp.org/mp-documents/OpenMP-4.0-C.pdf
Extra material
Invalid for Loop

```c
#include <stdio.h>
#ifdef __OPENMP
#include <omp.h>
#endif

int search(int key, int N, int* A) {
    #pragma omp parallel for
    for (int i = 0; i < N; i++)
        if (A[i] == key) return i;
    return -1;
}

int main() {
#ifdef __OPENMP
    omp_set_num_threads(4);
#endif
    int N = 10;
    int A[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int key = 5;

    int index = search(key, N, A);
    printf("key found at index = %d\n", index);
}
```
Data Affinity Example

```fortran
!$OMP DO PRIVATE(i)
do j=1,n
  do i=1,n
    a(i,j)= i+j
  end do
end do
!$OMP DO SCHEDULE(STATIC,16)
!$OMP& PRIVATE(i)
do j=1,n
  do i=1,j
    b(j)=b(j)+a(i,j)
  end do
end do
```

- 1st 16 j's OK – rest are cache misses!

Credit: Alan Real
Dropped material
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);
        }
    }
}

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

- UMA (SMP)
- ccNUMA

[Dirk Schmidl, Christian Terboven]→