Principles of High Performance Computing (ICS 632)

Shared-Memory Programming
with Threads

Shared memory programming

- The "easiest" form of parallel programming
- Can be used to parallelize a sequential code in an incremental way:
  - take a sequential code
  - parallelize a small section
  - check that it works
  - check that it speeds things up a bit
  - move on to another section
- We will see that parallelizing a program for distributed memory is far from trivial and requires much more effort
  - But it is necessary to scale to large numbers of processors
  - Remember that almost everybody would prefer a shared-memory machine to a distributed-memory machine. The problem is that shared-memory machines do not scale well within reasonable cost (if at all possible).

What is a thread?

- A thread is a stream of instructions that can be scheduled as an independent unit.
- A process is created by an operating system
  - contains information about resources
    - process id, file descriptors, ...
  - contains information on the execution state
    - program counter, stack, ...
- The concept of a thread requires that we make a separation between these two kinds of information in a process
  - resources available to the entire process
    - program instructions, global data, working directory
  - schedulable entities
    - program counters and stacks.
- A thread is an entity within a process which consists of the schedulable part of the process.

Parallelism with Threads

- Create threads within a process
- Each thread does something (hopefully) useful
- Threads may be working truly concurrently
  - Multi-processor
  - Multi-core
- Or just pseudo-concurrently
  - Single-proc, single-core

Example

- Say I want to compute the sum of two arrays
- I can just create N threads, each of which sums 1/Nth of both arrays and then combine their results
- I can also create N threads that each increment some sum variable element-by-element, but then I've got to make sure they don't step on each other's toes
- The first version is a bit less "shared-memory", but is probably more efficient

Multi-threading issues

- There are really two main issues when writing multi-threaded code:
- Issue #1: Load Balancing
  - Make sure that no processors/cores is left idle when it could be doing useful work
  - We will talk about this a lot throughout the semester as it arises in all forms of parallel computing
- Issue #2: Correct access to shared variables
  - Implemented via mutual exclusion: create sections of code that only a single thread can be in at a time
    - Called "critical sections"
    - Classical variable update example
  - Done via "locks" and "unlocks"
    - Warning: locks are NOT on variables, but on sections of code

User-level / Kernel-level

- User-level threads: Many-to-one thread mapping
  - Implemented by user-level runtime libraries
    - Create, schedule, synchronize threads at user-level
  - OS is not aware of user-level threads
    - OS thinks each process contains only a single thread of control
- Advantages
  - Does not require OS support; Portable
  - Can tune scheduling policy to meet application demands
  - Lower overhead thread operations since no system calls
- Disadvantages
  - Cannot leverage multiprocessors
  - Entire process blocks when one thread blocks

User-level / Kernel-level

- Kernel-level threads: One-to-one thread mapping
  - OS provides each user-level thread with a kernel thread
  - Each kernel thread scheduled independently
  - Thread operations (creation, scheduling, synchronization) performed by OS
- Advantages
  - Each kernel-level thread can run in parallel on a multiprocessor
  - When one thread blocks, other threads from process can be scheduled
- Disadvantages
  - Higher overhead for thread operations
  - OS must scale well with increasing number of threads

Threads in Practice

- Pthreads
  - Popular C library
  - Flexible
  - Requires a fair amount of work
- OpenMP
  - Standard for multi-threading for high-performance computing
  - More rigid than pthreads
  - Requires very little work
  - Most typical for HPC programming
- Java Threads
  - We won't talk about these (see ICS432)

Pthreads: POSIX Threads

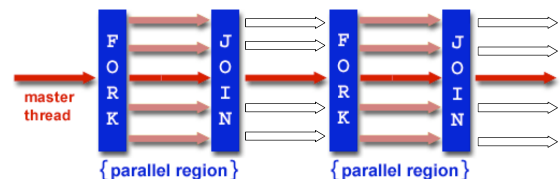
- Pthreads is a standard set of C library functions for multithreaded programming
  - IEEE Portable Operating System Interface, POSIX, section 1003.1 standard, 1995
- Pthread Library (60+ functions)
  - Thread management: create, exit, detach, join, . . .
  - Thread cancellation
  - Mutex locks: init, destroy, lock, unlock, . . .
  - Condition variables: init, destroy, wait, timed wait, . . .
  - . . .
- We won't talk about it in this class, but if you have no idea what they are you should really find out
  - They're not used a lot in traditional HPC programming, but they are extremely important for systems work

Threads with OpenMP

- Goal: make shared memory programming easy (or at least easier than with pthread)
- How?
  - A library with some simple functions
  - The definition of a few C pragmas
    - pragmas are a way to make the language extensible
    - provide an easy way to give hints/information to a compiler
  - A compiler (to generate pthread code!)

Fork-Join Model

- Program begins with a Master thread
- **Fork:** Teams of threads created at times during execution
- **Join:** Threads in the team synchronize (barrier) and only the master thread continues execution



OpenMP and #pragma

- One needs to specify blocks of code that are executed in parallel
- For example, *a parallel section*:

```
#pragma omp parallel [clauses]
```

  - Defines a section of the code that will be executed in parallel
  - The “clauses” specify many things including what happens to variables
  - All threads in the section execute the same code

OpenMP Compiler

- There are several free OpenMP “compiler”
  - Really more like source-to-source translators
- The OpenMP compiler on our cluster is called **`ompicc`**
  - **`/usr/local/bin/ompicc`**
- Add it to your path
- You can use it just like gcc
  - all the options should work
  - but compilation error messages may be different (meaning worse)

First “Hello World” example

```
#include <omp.h>
int main(){
print("Start\n");
#pragma omp parallel
 { // note the {
   printf("Hello World\n");
 } // note the }
/* Resume Serial Code */
 printf("Done\n");
}

% my_program
Start
Hello World
Hello World
Hello World
Done
```

First “Hello World” example

```
#include <omp.h>
int main(){
print("Start\n");
#pragma omp parallel
 {
   printf("Hello World\n");
 }
/* Resume Serial Code */
 printf("Done\n");
}

% my_program
Start
Hello World
Hello World
Hello World
Done
```

- Questions
  - How many threads?
  - This is not useful because all threads do exactly the same thing
  - Conditional compilation?

How Many Threads?

- Set via an environment variable

  `setenv OMP_NUM_THREADS 8`
  - Bounds the maximum number of threads
- Set via the OpenMP API

```
void omp_set_num_threads(int number);
int omp_get_num_threads();
```

- Typically, a function of the number of processors available
  - We often take the number of threads identical to the number of processors/cores

How Many Threads

- On our cluster with our compiler you must have the **`OMP_NUM_THREADS`** env. variable set to some high value (e.g., 16)
  - Otherwise you’ll never be able to get more than a few threads, even my called omp_set_num_threads()
- You’ll have to use qsub -V to make sure that your environment variables are seen on the compute nodes when you submit a job, and not just the front end
  - It may be a good idea for you to alias “qsub” to “qsub -V”

Threads Doing Different Things

```
#include <omp.h>
int main()  {
  int iam =0, np = 1;
#pragma omp parallel private(iam, np)
  {
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
    printf("Hello from thread %d out of %d threads\n", iam,
   np);
  }
}
```

```
% setenv OMP_NUM_THREADS 3
% my_program
Hello from thread 0 out of 3
Hello from thread 1 out of 3
Hello from thread 2 our of 3
```

Conditional Compilation

- The _OPENMP variable is defined if the code is compiled with OpenMP

```
#ifdef _OPENMP
#include <omp.h>
#ifdef _OPENMP
int main()  {
  int iam = 0, np = 1;
#pragma omp parallel private(iam, np)
  {
#ifdef _OPENMP
    np = omp_get_num_threads();
    iam = omp_get_thread_num();
#endif
    printf("Hello from thread %d out of %d threads\n", iam, np);
  }
}
```

- This code will work serially!

Data Scoping and Clauses

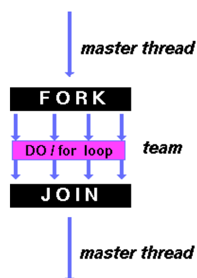
- Shared: all threads access the **single copy** of the variable, created in the master thread
  - it is the responsibility of the programmer to ensure that it is shared appropriately
- Private: a **volatile** copy of the variable is created for **each thread**, and discarded at the end of the parallel region (but for the master)
- There are other variations
  - firstprivate: initialization from the master's copy
  - lastprivate: the master gets the last value updated by the last thread to do an update
  - and several others

  (Look in the on-line material if you're interested)

Work Sharing directives

- We have seen the concept of a parallel region, which is a brute-force SPMD directive
- Work Sharing directives make it possible to have threads "share work" *within a parallel region*.
  - For Loop
  - Sections
  - Single

For Loops



- Share iterations of the loop across threads
- Represents a type of "data parallelism"
  - do the same operation on pieces of the same big piece of data
- Program correctness must NOT depend on which thread executes which iteration
  - No ordering!

For Loop Example

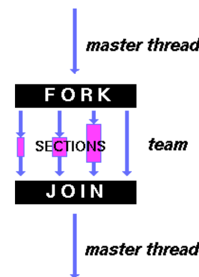
```
#include <omp.h>
#define N 1000
main ()  {
  int i, chunk;float a[N], b[N], c[N];
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i)
  {
  #pragma omp for schedule(dynamic)
  for (i=0; i < N; i++)
    c[i] = a[i] + b[i];
  }  /* end of parallel section - threads are
              synchronized*/
}
```

For Loop and "nowait"

- With "nowait", threads do not synchronize at the end of the loop
  - i.e., threads may exit the #pragma omp for at different times

```
#pragma omp parallel shared(a,b,c) private(i)
  {
  #pragma omp for schedule(dynamic) nowait
    for (i=0; i < N; i++) {
      // do some work
    }
    // Threads get here at different times
  }
```

Sections



- Breaks work into separate sections
- Each section is executed by a thread
- Can be used to implement "task parallelism"
  - do different things on different pieces of data
- If more threads than sections, then some are idle
- If fewer threads than sections, then some sections are seralized

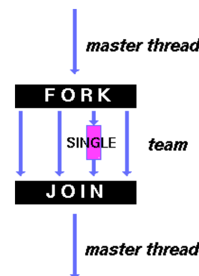
Section Example

```
#include <omp.h>
#define N 1000
main (){
  int i;float a[N], b[N], c[N];
  for (i=0; i < N; i++)
    a[i] = b[i] = i * 1.0;
#pragma omp parallel shared(a,b,c) private(i)
  {
  #pragma omp sections
    {
    #pragma omp section
      {
      for (i=0; i < N/2; i++)
        c[i] = a[i] + b[i];
      }
    #pragma omp section
      {
      for (i=N/2; i < N; i++)
        c[i] = a[i] + b[i];
      }
    }  /* end of sections */
  }  /* end of parallel section */
}
```

Section #1

Section #2

Single



- Serializes a section of code within a parallel region
- Sometimes more convenient than terminating a parallel region and starting it later
  - especially because variables are already shared/private, etc.
- Typically used to serialize a small section of the code that's not thread safe
  - e.g., I/O

Combined Directives

- It is cumbersome to create a parallel region and *then* create a parallel for loop, or sections, just to terminate the parallel region
- Therefore OpenMP provides a way to do both at the same time
  - #pragma omp parallel for
  - #pragma opm parallel sections

Synchronization and Sharing

- When variables are shared among threads, OpenMP provides tools to make sure that the sharing is correct
- Why could things be unsafe?

```
int x = 0;
#pragma omp parallel sections shared(x)
  {
  #pragma omp section
  x = x + 1
  #pragma omp section
  x = x + 2
  }
```

Synchronization directive

- #pragma omp master
  - Creates a region that only the master executes
- #pragma omp critical
  - Creates a critical section
- #pragma omp barrier
  - Creates a “barrier”
- #pragma omp atomic
  - Create a “mini” critical section

Critical Section

```
#pragma omp parallel for \
            shared(sum)
  for(i = 0; i < n; i++){
    value = f(a[i]);
    #pragma omp critical
    {
      sum = sum + value;
    }
  }
```

Barrier

```
if (x == 2) {
  #pragma omp barrier
}
```

- All threads in the current parallel section will synchronize
  - they will all wait for each other at this instruction
- Must appear within a basic block

Atomic

```
#pragma omp atomic
        i++;
```

- Only for some expressions
  - x = expr   (no mutual exclusion on expr evaluation)
  - x++
  - ++x
  - x--
  - --x
- Is about atomic access to a memory location
- Some implementations will just replace **atomic** by **critical** and create a basic blocks
- But some may take advantage of cool hardware instructions that work atomically

Scheduling

- When I talked about the parallel for loops, I didn’t say how the iterations were shared among threads
- Question: I have 100 iterations. I have 5 threads. Which thread does which iteration?
- OpenMP provides many options to do this
- Choice #1: Chunk size
  - a way to group iterations togethers
  - e.g., chunk size = 2 means that iterations are grouped 2 by 2
  - allows to avoid prohibitive overhead in some situations
- Choice #2: Scheduling Policy

Loop Scheduling in OpenMP

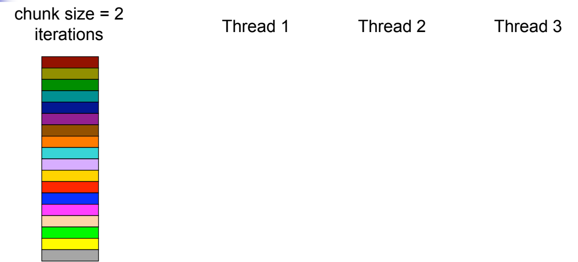
- static:
  - Iterations are divided into pieces of a size specified by chunk.
  - The pieces are statically assigned to threads in the team in a roundrobin fashion in the order of the thread number.
- dynamic:
  - Iterations are broken into pieces of a size specified by chunk.
  - As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations.
- guided:
  - The chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space.
  - chunk specifies the smallest piece (except possibly the last).
- Default schedule: implementation dependent.

Example

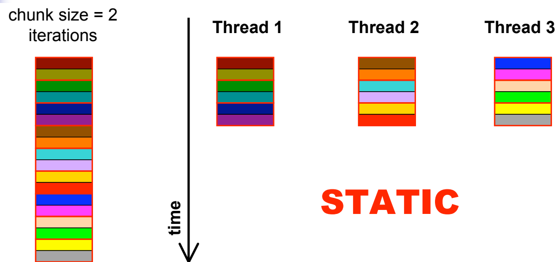
```
int chunk = 2

#pragma omp parallel for \
       shared(a,b,c,chunk) \
       private(i) \
       schedule(static,chunk)
  for (i=0; i < n; i++)
    c[i] = a[i] + b[i];}
```

OpenMP Scheduling



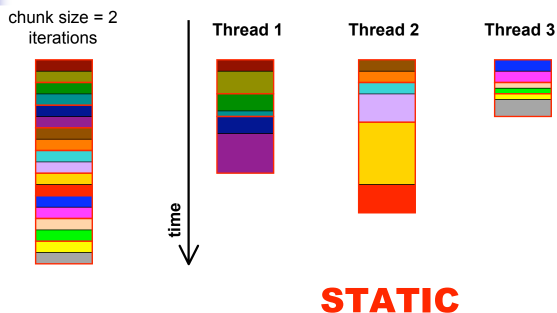
OpenMP Scheduling



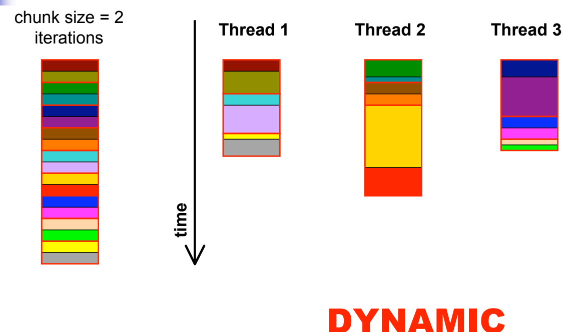
So, isn't static optimal?

- The problem is that in may cases the iterations are not identical
  - Some iterations take longer to compute than other
- Example #1
  - Each iteration is a rendering of a movie's frame
  - More complex frames require more work
- Example #2
  - Each iteration is a "google search"
  - Some searches are easy
  - Some searches are hard
- In such cases, load unbalance arises
  - which we know is bad

OpenMP Scheduling



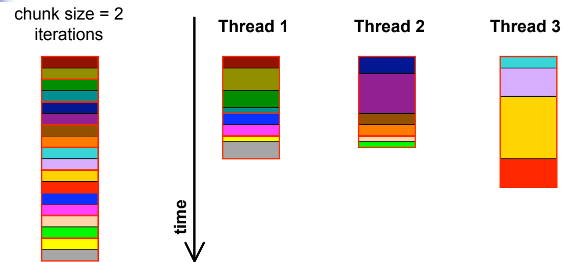
OpenMP Scheduling



So isn't dynamic optimal?

- One thing we haven't talked much about is the overhead
- Dynamic scheduling with small chunks causes more overhead than static scheduling
  - In the static case, one can compute what each thread does at the beginning of the loop and then let the threads proceed unhindered
  - In the dynamic case, there needs to be some type of communication: "I am done with my 2 iterations, which ones do I do next?"
    - Can be implemented in a variety of ways internally
- Using dynamic scheduling with a large chunk size leads to lower overhead, but defeats the purpose
  - with fewer chunks, load-balancing is harder
- Guided Scheduling: best of both worlds
  - start with large chunks, ends with small ones

OpenMP Scheduling



Guided

3 chunks of size 4
2 chunks of size 2

What should I do?

- Pick a reasonable chunk size
- Use static if computation is evenly spread among iterations
- Otherwise probably use guided

How does OpenMP work?

- The pragmas allow OpenMP to build some notion of structure of the code
- And then, OpenMP generates pthread code!!
  - You can see this by running the `nm` command on your executable
- OpenMP hides a lot of the complexity
- But it doesn't have all the flexibility
- The two are used in different domains
  - OpenMP: "scientific applications"
  - Pthreads: "system" applications
- But this distinction is really arbitrary IMHO

More OpenMP Information

- OpenMP Homepage: http://www.openmp.org/
- On-line OpenMP Tutorial: http://www.llnl.gov/computing/tutorials/openMP/

Lessons

- Although we have only scratched the surface of parallel computing, we have already encountered a few fundamental concepts
- Load-balancing is good
- Overhead is bad
- The two often pose a difficult trade-off
  - Achieving great load-balancing can often only be done with high overhead, which puts us back where we started
- We will see this trade-off over and over in many different contexts