Lecture 5:
Data and Task Parallelism, cont.
Data Parallelism in OpenMP

Mary Hall
September 7, 2010

Homework 2, cont
Problem 2 (based on #2 in text on p. 59), cont.:
Now provide a bound for the memory cost for a particular thread predicted by CTA for the Try4 algorithm of Fig. 114 on p. 23 (or Try3 assuming each element is placed on a separate cache line).

Problem 3:
For these examples, how is algorithm selection impacted by the value of NTPT?

Problem 4 (in general, not specific to this problem):
How is algorithm selection impacted by the value of E?

Today’s Lecture
• Review Data and Task Parallelism
• Brief Overview of POSIX Threads
• Data Parallelism in OpenMP
  - Expressing Parallel Loops
  - Parallel Regions (SPMD)
  - Scheduling Loops
  - Synchronization
• Sources of material:
  - Jim Demmel and Kathy Yelick, UCB
  - Allan Snavely, SDSC
  - Larry Snyder, Univ. of Washington
Definitions of Data and Task Parallelism

- **Data parallel computation:**
  - Perform the same operation to different items of data at the same time; the parallelism grows with the size of the data.

- **Task parallel computation:**
  - Perform distinct computations -- or tasks -- at the same time; with the number of tasks fixed, the parallelism is not scalable.

- **Summary**
  - Mostly we will study data parallelism in this class.
  - Data parallelism facilitates very high speedups; and scaling to supercomputers.
  - Hybrid (mixing of the two) is increasingly common.

Examples of Task and Data Parallelism

- Looking for all the appearances of "University of Utah" on the world-wide web.
- A series of signal processing filters applied to an incoming signal.
- Same signal processing filters applied to a large known signal.
- Climate model from Lecture 1.

Review: Predominant Parallel Control Mechanisms

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiple Instruction, Multiple Data (MIMD)</td>
<td>Multiple threads of control, processors periodically synch</td>
<td>Parallel loop:forall (i=0; i&lt;n; i++)</td>
</tr>
<tr>
<td>Single Program, Multiple Data (SPMD)</td>
<td>Multiple threads of control, but each processor executes same code</td>
<td>Processor-specific code: if ($myid == 0) {</td>
</tr>
</tbody>
</table>

Programming with Threads

- **PTHREADS** is the Posix Standard
  - Solaris threads are very similar
  - Relatively low level
  - Portable but possibly slow

- **OpenMP** is newer standard
  - Support for scientific programming on shared memory architectures

- **P4 (Parmacs)** is another portable package
  - Higher level than Pthreads
Overview of POSIX Threads

• POSIX: Portable Operating System Interface for UNIX
  - Interface to Operating System utilities
• PThreads: The POSIX threading interface
  - System calls to create and synchronize threads
  - Should be relatively uniform across UNIX-like OS platforms
• PThreads contain support for
  - Creating parallelism
  - Synchronizing
  - No explicit support for communication, because shared memory is implicit; a pointer to shared data is passed to a thread

Forking Posix Threads

Signature:
  int pthread_create(pthread_t *,
                  const pthread_attr_t *,
                  void * (*)(void *),
                  void *);

Example call:
  errcode = pthread_create(&thread_id;
                          &thread_attribute
                          &thread_fun; &fun_arg);

• thread_id is the thread id or handle (used to halt, etc.)
• thread_attribute various attributes
  - standard default values obtained by passing a NULL pointer
• thread_fun the function to be run (takes and returns void*)
• fun_arg an argument can be passed to thread_fun when it starts
• errcode will be set nonzero if the create operation fails

Simple Threading Example

```c
void* SayHello(void *foo) {
    printf( "Hello, world!\n" );
    return NULL;
}

int main() {
    pthread_t threads[16];
    int tn;
    for(tn=0; tn<16; tn++) {
        pthread_create(&threads[tn], NULL, SayHello, NULL);
    }
    for(tn=0; tn<16 ; tn++) {
        pthread_join(threads[tn], NULL);
    }
    return 0;
}
```

Compile using gcc –lpthread

But overhead of thread creation is nontrivial

SayHello should have a significant amount of work

Shared Data and Threads

• Variables declared outside of main are shared
• Object allocated on the heap may be shared (if pointer is passed)
• Variables on the stack are private: passing pointer to these around to other threads can cause problems
• Often done by creating a large "thread data" struct
  - Passed into all threads as argument
  - Simple example:
    ```c
    char *message = "Hello World!\n";
    pthread_create( thread1,
                    NULL,
                    (void*)print_fun,
                    (void*) message);
    ```
Posix Thread Example

```c
#include <pthread.h>
void print_fun( void *message ) {
    printf("%s \n", message);
}

main() {
    pthread_t thread1, thread2;
    char *message1 = "Hello";
    char *message2 = "World";
    pthread_create(&thread1, NULL, (void*)print_fun, (void*)message1);
    pthread_create(&thread2, NULL, (void*)print_fun, (void*)message2);
    return(0);
}
```

Compile using gcc -lpthread

Note: There is a race condition in the print statements

Explicit Synchronization: Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):
  ```c
  pthread_barrier_t b;
  pthread_barrier_init(&b, NULL, 3);
  
  The second argument specifies an object attribute; using NULL yields the default attributes.
  ```

- To wait at a barrier, a process executes:
  ```c
  pthread_barrier_wait(&b);
  ```

- This barrier could have been statically initialized by assigning an initial value created using the macro
  `PTHREAD_BARRIER_INITIALIZER(3)`.

Mutexes (aka Locks) in POSIX Threads

- To create a mutex:
  ```c
  #include <pthread.h>
  pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
  pthread_mutex_init(&amutex, NULL);
  ```

- To use it:
  ```c
  int pthread_mutex_lock(amutex);
  int pthread_mutex_unlock(amuex);
  ```

- To deallocate a mutex:
  ```c
  int pthread_mutex_destroy(pthread_mutex_t amutex);
  ```

- Multiple mutexes may be held, but can lead to deadlock:
  ```c
  thread1           thread2
  lock(a)           lock(b)
  lock(b)           lock(a)
  ```

Summary of Programming with Threads

- POSIX Threads are based on OS features
  - Can be used from multiple languages (need appropriate header)
  - Familiar language for most programmers
  - Ability to shared data is convenient

- Pitfalls
  - Data race bugs are very nasty to find because they can be intermittent
  - Deadlocks are usually easier, but can also be intermittent

- OpenMP is commonly used today as a simpler alternative, but it is more restrictive
OpenMP Motivation

- Thread libraries are hard to use
  - P-Threads/Solaris threads have many library calls for initialization, synchronization, thread creation, condition variables, etc.
  - Programmer must code with multiple threads in mind
- Synchronization between threads introduces a new dimension of program correctness
- Wouldn’t it be nice to write serial programs and somehow parallelize them “automatically”? 
  - OpenMP can parallelize many serial programs with relatively few annotations that specify parallelism and independence.
  - It is not automatic: you can still make errors in your annotations.

A Programmer’s View of OpenMP

- OpenMP is a portable, threaded, shared-memory programming specification with “light” syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into serial regions and parallel regions, rather than concurrently-executing threads.
  - Hide stack management
  - Provide synchronization constructs
- OpenMP will not:
  - Parallelize automatically
  - Guarantee speedup
  - Provide freedom from data races

OpenMP: Prevailing Shared Memory Programming Approach

- Model for parallel programming
- Shared-memory parallelism
- Portable across shared-memory architectures
- Scalable
- Incremental parallelization
- Compiler based
- Extensions to existing programming languages (Fortran, C and C++)
  - mainly by directives
  - a few library routines

OpenMP Data Parallel Construct: Parallel Loop

- All pragmas begin: #pragma
- Compiler calculates loop bounds for each thread directly from serial source (computation decomposition)
- Compiler also manages data partitioning of Res
- Synchronization also automatic (barrier)

<table>
<thead>
<tr>
<th>Serial Program:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void main() {</code></td>
</tr>
<tr>
<td><code>double Res[100];</code></td>
</tr>
<tr>
<td><code>for(int i=0;i&lt;100;i++</code></td>
</tr>
<tr>
<td><code>{</code></td>
</tr>
<tr>
<td><code>do_bugs_comp[Res][i]</code></td>
</tr>
<tr>
<td><code>}</code></td>
</tr>
<tr>
<td><code>}</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Parallel Program:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void main()</code></td>
</tr>
<tr>
<td><code>{</code></td>
</tr>
<tr>
<td><code>double Res[100];</code></td>
</tr>
<tr>
<td><code>#pragma omp parallel for</code></td>
</tr>
<tr>
<td><code>for(int i=0;i&lt;100;i++</code></td>
</tr>
<tr>
<td><code>{</code></td>
</tr>
<tr>
<td><code>do_bugs_comp[Res][i]</code></td>
</tr>
<tr>
<td><code>}</code></td>
</tr>
<tr>
<td><code>}</code></td>
</tr>
</tbody>
</table>
OpenMP Execution Model

- Fork-join model of parallel execution
- Begin execution as a single process (master thread)
- Start of a parallel construct:
  - Master thread creates team of threads
- Completion of a parallel construct:
  - Threads in the team synchronize -- implicit barrier
- Only master thread continues execution
- Implementation optimization:
  - Worker threads spin waiting on next fork

Count 3s Example? (see textbook)

What do we need to worry about?

OpenMP directive format C
(also Fortran and C++ bindings)

- Pragmas, format
  
  ```c
  #pragma omp directive_name [ clause [ clause ] ... ]
  ```

- Conditional compilation
  
  ```c
  #ifdef _OPENMP
  block.
  #ifdef printf("%d avail.processors\n",omp_get_num_procs());
  #endif
  ```

- Case sensitive

- Include file for library routines
  
  ```c
  #ifdef _OPENMP
  #include <omp.h>
  #endif
  ```
Limitations and Semantics

- Not all "element-wise" loops can be parallelized

```c
#pragma omp parallel for
for (i=0; i < numPixels; i++) {}
```

- Loop index: signed integer
- Termination Test: <,<=,>,=> with loop invariant int
- Incr/Decr by loop invariant int; change each iteration
- Count up for <, count down for >
- Basic block body: no control in/out except at top

- Threads are created and iterations divvied up; requirements ensure iteration count is predictable

OpenMP Synchronization

- Implicit barrier
  - At beginning and end of parallel constructs
  - At end of all other control constructs
  - Implicit synchronization can be removed with nowait clause

- Explicit synchronization
  - critical
  - atomic

OpenMP Reductions

- OpenMP has reduce

```c
sum = 0;
#pragma omp parallel for reduction(+sum)
for (i=0; i < 100; i++) {
    sum += array[i];
}
```

- Reduce ops and init() values:
  - `+`     bitwise `&`   ~0       logical `&`   1
  - `-`     bitwise `|`   0       logical `|`   0
  - `*`     bitwise `^`   0

OpenMP parallel region construct

- Block of code to be executed by multiple threads in parallel
- Each thread executes the same code redundantly (SPMD)
  - Work within work-sharing constructs is distributed among the threads in a team
- Example with C/C++ syntax

```c
#pragma omp parallel [clause [clause] ...] new-line structured-block
```

- clause can include the following:
  - private (list)
  - shared (list)
Programming Model – Loop Scheduling

- schedule clause determines how loop iterations are divided among the thread team
  - static([chunk]) divides iterations statically between threads
    - Each thread receives [chunk] iterations, rounding as necessary to account for all iterations
    - Default [chunk] is ceil( # iterations / # threads )
  - dynamic([chunk]) allocates [chunk] iterations per thread, allocating an additional [chunk] iterations when a thread finishes
    - Forms a logical work queue, consisting of all loop iterations
    - Default [chunk] is 1
  - guided([chunk]) allocates dynamically, but [chunk] is exponentially reduced with each allocation

OpenMP critical directive

- Enclosed code
  
  - executed by all threads, but
  
  - restricted to only one thread at a time

  #pragma omp critical [( name )] new-line
  structured-block

- A thread waits at the beginning of a critical region until no other thread in the team is executing a critical region with the same name.

- All unnamed critical directives map to the same unspecified name.

Variation: OpenMP parallel and for directives

Syntax:

```c
#pragma omp parallel private(f)
{ f=7; 
#pragma omp for schedule(static)
for (i=0; i<20; i++)
a[i] = b[i] + f * (i+1);
} /*omp end parallel */
```

```c
#pragma omp for
for (i=0; i<20; i++)
a[i] = b[i] + f * (i+1); 
```

```c
```

loop scheduling

static       dynamic(1)       guided(1)

09/07/2010 CS4961
**Programming Model – Data Sharing**

- Parallel programs often employ two types of data:
  - Shared data, visible to all threads, similarly named
  - Private data, visible to a single thread (often stack-allocated)

- **PThreads**:
  - Global-scoped variables are shared
  - Stack-allocated variables are private

- **OpenMP**:
  - `shared` variables are shared
  - `private` variables are private
  - Default is `shared`
  - Loop index is `private`

```c
void* foo(void* bar) {
    int tid;
    #pragma omp parallel \
    shared ( bigdata ) \
    private ( tid ) 
    { /* Calc. here */ }
}
```

**OpenMP environment variables**

- **OMP_NUM_THREADS**
  - Sets the number of threads to use during execution
  - When dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use
  - For example,
    ```
    setenv OMP_NUM_THREADS 16 [csh, tcsh]
    export OMP_NUM_THREADS=16 [sh, ksh, bash]
    ```

- **OMP_SCHEDULE**
  - Applies only to do/for and parallel do/for directives that have the schedule type `RUNTIME`
  - Sets schedule type and chunk size for all such loops
  - For example,
    ```
    setenv OMP_SCHEDULE GUIDED,4 [csh, tcsh]
    export OMP_SCHEDULE= GUIDED,4 [sh, ksh, bash]
    ```

**OpenMP runtime library, Query Functions**

- `omp_get_num_threads(void);`
  - Returns the number of threads currently in the team executing the parallel region from which it is called

- `omp_get_thread_num(void);`
  - Returns the thread number, within the team, that lies between 0 and `omp_get_num_threads()-1`, inclusive. The master thread of the team is thread 0

**Summary of Lecture**

- **OpenMP**, data-parallel constructs only
  - Task-parallel constructs later

- **What’s good?**
  - Small changes are required to produce a parallel program from sequential (parallel formulation)
  - Avoid having to express low-level mapping details
  - Portable and scalable, correct on 1 processor

- **What is missing?**
  - Not completely natural if want to write a parallel code from scratch
  - Not always possible to express certain common parallel constructs
  - Locality management
  - Control of performance