CS4961 Parallel Programming

Lecture 8: Introduction to Threads and Data Parallelism in OpenMP

Mary Hall
September 17, 2009

Administrative

- Programming assignment 1 is posted (after class)
- Due, Tuesday, September 24 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 prog1 <gzipped tar file>"
- Mailing list set up: cs4961@list.eng.utah.edu

Project 1 Assignment

- **PART I**
- Each of the files t2.c, t3.c, t4.c and t5.c from the website cannot be vectorized by the ICC compiler. Your assignment is to produce the equivalent but vectorizable nt2.c, nt3.c, nt4.c and n5t.c. To determine whether or not the compiler has vectorized a loop in the code, look at the output of the compiler that results from the flag –vec-report3. If it says: "remark: LOOP WAS VECTORIZED.", then you have succeeded! Hints: There are several reasons why the above examples cannot be vectorized. In some cases, the compiler is concerned about the efficiency of the vectorized code. This may be because of the cost of alignment, concerns about exceeding the register capacity (there are only 8 128-bit registers on most SSE3-supporting platforms!) or the presence of data dependences. In other cases, there is concern about correctness, due to aliasing.

Project 1 Assignment, cont.

- **PART II**

The code example youwrite.c is simplified from a sparse matrix-vector multiply operation used in conjugate gradient. In the example, the compiler is able to generate vectorizable code, but it must deal with alignment in the inner loop. In this portion, we want you to be a replacement for the compiler and use the intrinsic operations found in Appendix C (pp. 832-844) of the document found at: http://developer.intel.com/design/pentiumii/manuals/243191.htm. The way this code will be graded is from looking at it (no running of the code this time around), so it would help if you use comments to describe the operations you are implementing. It would be a good idea to test the code you write, but you will need to generate some artificial input and compare the output of the vectorized code to that of the sequential version.

Things to Remember in this Assignment

- What are the barriers to “vectorization”?
  - Correctness
    - Dependences
  - Performance
    - Alignment
    - Control flow
    - Packing to get data contiguous

Today’s Lecture

- 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

Programming with Threads

Several Thread Libraries

- 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
  - http://www.netlib.org/p4/index.html

Overview of POSIX Threads

- POSIX: *Portable **O**perating **S**ystem **I**nterface for UNI**X***
  - 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
- errorcode will be set nonzero if the create operation fails

Simple Threading Example

```
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:

```
char *message = "Hello World!\n";
pthread_create( &thread1,
        NULL,
        (void*)&print_fun,
        (void*) message);
```

Posix Thread Example

```
#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

Synchronization: Creating and Initializing a Barrier

- To (dynamically) initialize a barrier, use code similar to this (which sets the number of threads to 3):

```
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:

```
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:

```
#include <pthread.h>
pthread_mutex_t amutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_init(&amutex, NULL);
```

- To use it:

```
int pthread_mutex_lock(amutex);
int pthread_mutex_unlock(amutex);
```

- To deallocate a mutex

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- Multiple mutexes may be held, but can lead to deadlock:

```
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 of program
  - 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

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

See http://www.openmp.org

A Programmer's View of OpenMP

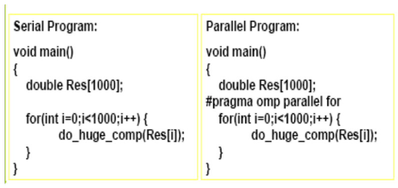
- OpenMP is a portable, threaded, shared-memory programming *specification* with "light" syntax
  - Exact behavior depends on OpenMP implementation!
  - Requires compiler support (C/C++ or Fortran)
- OpenMP will:
  - Allow a programmer to separate a program into *serial regions* and *parallel regions,*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

Open MP Example: Parallel Loop

- All pragmas begin: #pragma
- Example: Convert 32-bit RGB image to 8-bit gray scale
- ||ism is "element-wise" … for correctness, each element must be independent ***(work sharing)***
- Preprocessor calculates loop bounds for each thread directly from *serial* source

```
#pragma omp parallel for
for (i=0; i < numPixels; i++) {
   pGrayScaleBitmap[i] = (unsigned BYTE)
    (pRGBBitmap[i].red   * 0.299 +  pRGBBitmap[i].green *
    0.587 +  pRGBBitmap[i].blue  * 0.114);
}
```

Another OpenMP example



Serial Program:	Parallel Program:

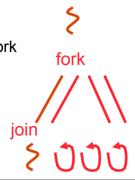
```
void main()
{
   double Res[1000];

   for(int i=0;i<1000;i++) {
        do_huge_comp(Res[i]);
   }
}
```
	```
	void main()
	{
	   double Res[1000];
	#pragma omp parallel for
	   for(int i=0;i<1000;i++) {
	        do_huge_comp(Res[i]);
	   }
	}
	```

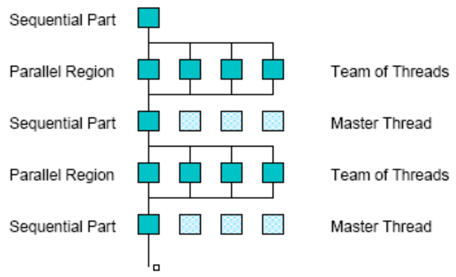
Pragmas as modest extension to existing language
- * Parallelism (SPMD, task and data parallel)
- * Data sharing (shared, private, reduction)
- * Work sharing or scheduling
- * Locks and critical sections

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



OpenMP Execution Model



OpenMP directive format C

- Pragmas, format

  `#pragma omp` directive_name [clause [clause] ...] new-line
- Conditional compilation

```
#ifdef _OPENMP
    block,
    e.g., printf("%d avail.processors\n",omp_get_num_procs());
#endif
```

- Case sensitive
- Include file for library routines

```
#ifdef _OPENMP
#include <omp.h>
#endif
```

OpenMP for directive

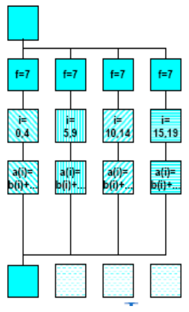
Syntax:

`#pragma omp for` [clause [clause] ...] new-line

for-loop

clause can be one of the following:

`shared (list)`
`private(` list`)`
`reduction(` operator`:` list`)`
`schedule(` type [, chunk] `)`
`nowait` (C/C++: on `#pragma omp for`)



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

Limitations and Semantics

- Not all "element-wise" loops can be ||ized

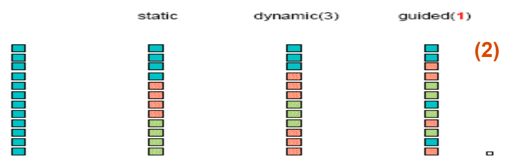
```
#pragma omp parallel for
for (i=0; i < numPixels; i++) {}
```

  - 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

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

Loop scheduling



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

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

  structured-block
- clause can include the following:

  `private` (list)

  `shared` (list)

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**

```
// shared, globals
int bigdata[1024];

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]
```

09/17/2010 CS4961

OpenMP runtime library, Query Functions

`omp_get_num_threads`:

Returns the number of threads currently in the team executing the parallel region from which it is called

```
int omp_get_num_threads(void);
```

`omp_get_thread_num:`

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`

```
int omp_get_thread_num(void);
```

09/17/2010 CS4961

OpenMp Reductions

- OpenMP has reduce

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

- Reduce ops and init() values:

+	0	bitwise &	~0	logical &	1
-	0	bitwise |	0	logical |	0
*	1	bitwise ^	0		

09/17/2010 CS4961

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 critical directive

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

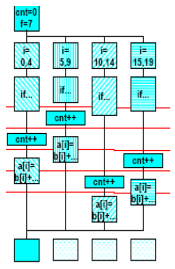
`#pragma omp critical [ ( name ) ]` new-line

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.

OpenMP critical

```
C / C++: cnt = 0;
f=7;
#pragma omp parallel
{
#pragma omp for
  for (i=0; i<20; i++) {
    if (b[i] == 0) {
#pragma omp critical
          cnt ++;
    } /* endif */
  a[i] = b[i] + f * (i+1);
  } /* end for */
} /*omp end parallel */
```



Summary of Lecture

- OpenMP, data-parallel constructs only
  - Task-parallel constructs next time
- What's good?
  - Small changes are required to produce a parallel program from sequential
  - Avoid having to express low-level mapping details
  - Portable and scalable, correct on 1 processor
- What is missing?
  - No scan
  - 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