CS4961 Parallel Programming

Lecture 9: Task Parallelism in OpenMP

Mary Hall
September 22, 2009

Administrative

- Programming assignment 1 is posted (after class)
- Due, Tuesday, September 22 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

Today's Lecture

- Go over questions on Proj1
- Review of OpenMP Data Parallelism
- Discussion of Task Parallelism in Open MP 2.x and 3.0
- Sources for Lecture:
  - OpenMP Tutorial by Ruud van der Pas
    http://openmp.org/mp-documents/ntu-vanderpas.pdf
  - OpenMP 3.0 specification (May 2008):
    http://www.openmp.org/mp-documents/spec30.pdf

Using Intel Software on VS2008

- File -> New Project (C++)
- Right Click Project or Project -> Intel Compiler -> Use Intel Compiler
- Project -> Properties -> C/C++ -> General -> Suppress Startup Banner = No
- Project -> Properties -> C/C++ -> Optimization -> Maximize Speed (/O2)
- Project -> Properties -> C/C++ -> Optimization -> Enable Intrinsic Functions (/Oi)
- Project -> Properties -> Code Generation -> Runtime Library -> Multithreaded DLL (/MD)
- Project -> Properties -> Code Generation -> Enable Enhanced Instruction Set = Streaming SIMD Extensions 3 (/arch:SSE3)
- Project -> Properties -> Command Line -> Additional Options -> Add /Qvec-report:3
- Click Apply
- Your command line options should look like

```
/c /O2 /Oi /D "WIN32" /D "_DEBUG" /D "_CONSOLE" /D
"_UNICODE" /D "UNICODE" /EHsc /MD /GS /arch:SSE3 /fp:fast /
Fo"Debug/" /W3 /ZI /Qvec-report:3
```

Project 1 Assignment

- **PART I**
- Each of the files t1.c, 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 nt1.c, 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.

Examples from Assignment

T2.c:

```
maxval = 0.;
for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
}
```

T3.c:

```
for (i=0; i<n; i++) {
    a[i] = b[i*4] + c[i];
}
```


Examples from Assignment

T4.c:

```
int start = rand();
for (j=start; j<n; j+=2) {
 for (i=start; i<n; i+=2) {
     a[i][j] = b[i][j] + c[i][j];
     a[i+1][j] = b[i+1][j] + c[i+1][j];

     a[i][j+1] = b[i][j+1] + c[i][j+1];
     a[i+1][j+1] = b[i+1][j+1] + c[i+1][j+1];
 }
}
```

T5.c:

```
for (i=0; i<n; i++) {
    t = t + abs(t-i);
    a[i] = t* b[i] + c[i];
}
```

Examples from Assignment

Youwrite.c:

```
for (i=0; i<n; i++) {
  scanf("%f %f %d\n", &a[i], &x[i], &colstr[i]);
}

for (i=0; i<n; i++) {
  t[i] = 0.0;
}

for (j=0; j<n; j++) {
  for (k = colstr[j]; k<colstr[j+1]-1; k++)
    t[k] = t[k] + a[k] * x[j];
}
}
```


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

OpenMP Summary

- Work sharing
  - parallel, parallel for, TBD
  - scheduling directives: static(CHUNK), dynamic(), guided()
- Data sharing
  - shared, private, reduction
- Environment variables
  - **OMP_NUM_THREADS, OMP_SET_DYNAMIC, OMP_NESTED, OMP_SCHEDULE**
- Library
  - E.g., omp_get_num_threads(), omp_get_thread_num()

Summary of Previous 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

Conditional Parallelization

if (scalar expression)

✓*Only execute in parallel if expression evaluates to true*
✓*Otherwise, execute serially*

```
#pragma omp parallel if (n > threshold) \
shared(n,x,y) private(i)   {
#pragma omp for
for (i=0; i<n; i++)
   x[i] += y[i];
} /*-- End of parallel region --*/
```

Review:
parallel
for
private
shared

Locks

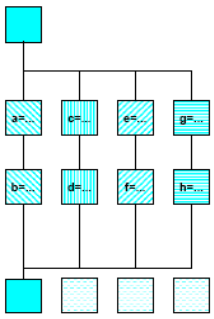
- Simple locks:
  - may not be locked if already in a locked state
- Nestable locks:
  - may be locked multiple times by the same thread before being unlocked

Simple locks
omp_init_lock
omp_destroy_lock
omp_set_lock
omp_unset_lock

Nestable locks
omp_init_nest_lock
omp_destroy_nest_lock
omp_set_nest_lock
omp_unset_nest_lock

OpenMP sections directive

```
#pragma omp parallel
{
#pragma omp sections
#pragma omp section
  {{ a=...;
  b=...; }
#pragma omp section
  { c=...;
  d=...; }
#pragma omp section
  { e=...;
  f=...; }
#pragma omp section
  { g=...;
  h=...; }
} /*omp end sections*/
} /*omp end parallel*/
```



Parallel Sections, Example

```
#pragma omp parallel default(none)\
shared(n,a,b,c,d) private(i)   {
  #pragma omp sections nowait     {
    #pragma omp section
      for (i=0; i<n; i++)
        d[i] = 1.0/c[i];
    #pragma omp section
      for (i=0; i<n-1; i++)
        b[i] = (a[i] + a[i+1])/2;
  } /*-- End of sections --*/
} /*-- End of parallel region
```


SINGLE and MASTER constructs

- Only one thread in team executes code enclosed
- Useful for things like I/O or initialization
- No implicit barrier on entry or exit

```
#pragma omp single {
   <code-block>
}
```

- Similarly, only master executes code

```
#pragma omp master {
   <code-block>
}
```


Motivating Example: Linked List Traversal

```
.....
while(my_pointer) {
    (void) do_independent_work (my_pointer);
    my_pointer = my_pointer->next ;
} // End of while loop
.....
```

- How to express with parallel for?
  - Must have fixed number of iterations
  - Loop-invariant loop condition and no early exits
- Convert to parallel for
  - A priori count number of iterations (if possible)

OpenMP 3.0: Tasks!

```
my_pointer = listhead;
#pragma omp parallel   {
  #pragma omp single nowait      {
    while(my_pointer) {
      #pragma omp task firstprivate(my_pointer)          {
        (void) do_independent_work (my_pointer);
      }
      my_pointer = my_pointer->next ;
    }
  } // End of single - no implied barrier (nowait)
} // End of parallel region - implied barrier here
```

firstprivate = private and copy initial value from global variable
lastprivate = private and copy back final value to global variable

Summary

- Completed coverage of OpenMP
  - Locks
  - Conditional execution
  - Single/Master
  - Task parallelism
    - Pre-3.0: parallel sections
    - OpenMP 3.0: tasks
- Next time:
  - OpenMP programming assignment