CS4961 Parallel Programming

Lecture 11: Thread Building Blocks, cont. and Reasoning about Performance

Mary Hall
September 29, 2009

Administrative

- Programming assignment 2 is posted (after class)
- Due, Thursday, October 8 before class
  - Use the "handin" program on the CADE machines
  - Use the following command:

    "handin cs4961 prog2 <gzipped tar file>"
- Mailing list set up: cs4961@list.eng.utah.edu

- Midterm Quiz on Oct. 8?
- Remote access to window machines?
  - term.coe.utah.edu does not support VS ☹

Today's Lecture

- Project 2
- Thread Building Blocks, cont.
- Ch. 3, Reasoning About Performance
- Sources for Lecture:
  - http://www.threadingbuildingblocks.org/
  - Tutorial:

    http://software.intel.com/sites/products/documentation/hpc/tbb/tutorial.pdf
  - Intel Academic Partners program (see other slides)

Project 2

- Part I Open MP

Problem 1 (Data Parallelism): The code from the last assignment models a sparse matrix vector multiply (updated in sparse_matvec.c). The matrix is sparse in that many of its elements are zero. Rather than representing all of these zeros which wastes storage, the code uses a representation called Compressed Row Storage (CRS), which only represents the nonzeros with auxiliary data structures to keep track of their location in the full array.

Given sparse_matvec.c, develop an OpenMP implementation of this code for 4 threads. You will also need to modify the initialization code as described below, and add timer functions. You will need to evaluate the three different scheduling mechanisms, static, dynamic and guided, and for two different chunk sizes of your choosing.

I have provided three input matrices, sm1.txt, sm2.txt2 and sm3.txt3, which were generated from the MatrixMarket (see http://math.nist.gov/MatrixMarket/). The format for these is a sorted coordinate representation (row, col, value) and will need to be converted to CRS. Measure the execution time for the sequential code and all three parallel versions, all three data set sizes and both chunk sizes.

You will turn in the code, and a brief README file with the 21 different timings and an explanation of which strategies performed best and why.

Part I, Problem 1

Read first non-comment line of input:
numrows, numcols, numelts
Allocate memory for a, t, x, rowstr, colind
Initialize a, rowstr and colind

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

sm1.txt

5 5 10
1 1 8.7567915491768E-1
1 2 7.0294465771411E-1
2 3 4.9541022395547E-1
2 5 6.3917764724488E-1
3 1 7.7804386900087E-1
3 4 4.3333577730521E-1
3 5 4.1076157239530E-2
4 4 1.5584897473534E-1
5 2 5.1359919564256E-1
5 3 1.0235676217063E-1

a:
.87 .70 .49 .63 .77 .43 .04 .15 .51 .10

Colind:
1 2 3 5 1 4 5 4 2 3

rowstr:
0 2 4 7 8

Project 2, cont.

- Part I Open MP, cont.

Problem 2 (Task Parallelism): Producer-consumer codes represent a common form of a task parallelism where one task is "producing" values that another thread "consumes". It is often used with a stream of data to implement pipeline parallelism.

The program prodcons.c implements a producer/consumer sequential application where the producer is generating array elements, and the consumer is summing up their values. You should use OpenMP parallel sections to implement this producer-consumer model. You will also need a shared queue between the producer and consumer tasks to hold partial data, and synchronization to control access to the queue. Create two parallel versions: producing/consuming one value at a time, and producing/consuming 128 values at a time.

Measure performance of the sequential code and the two parallel implementations and include these measurements in your README file.

Part I, Problem 2

```
#define N 166144
A = (double *)malloc(N*sizeof(double));
runtime = omp_get_wtime(); // need to replace timer
printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
fill_rand(N, A);        // Producer: fill an array of data
sum = Sum_array(N, A);  // Consumer: sum the array
runtime = omp_get_wtime(); // need to replace timer
printf(" In %lf seconds, The sum is %lf \n",runtime,sum);
```

What is needed for this one? (Hint: keep it simple)

Project 2, cont.

- Part II Thread Building Blocks

As an Academic Alliance member, we have access to Intel assignments for ThreadBuildingBlocks. We will use the assignments from Intel, with provided code that needs to be modified to use TBB constructs. You will turn in just your solution code.

Problem 3 (Problem 1 in TBB.doc, Using parallel_for)

Summary: Parallelize "mxm_serial.cpp"

Problem 4 (Problem 3 in TBB.doc, Using recursive tasks)

Summary: Modify implementation in rec_main.cpp

All relevant files prepended with rec_ to avoid conflict.

Problem 5 (Problem 4 in TBB.doc, Using the concurrent_hash_map container)

Summary: Modify implementation in chm_main.cpp

All relevant files prepended with chm_ to avoid conflict.

Part II, Problem 1 (see Tutorial, p. 10, sec 3.2)

Mxm_serial.cpp

```
void mxm( float c[N][N], float a[N][N], float b[N][N] ) {
  for( int i = 0; i < N; ++i ) {
    for( int j=0; j<N; ++j ) {
      float sum = 0;
          for( int k=0; k<N; ++k ) {
        sum += a[i][k]*b[k][j];
          }
      c[i][j] = sum;
    }
  }
}

// Rewrite this function to make use of TBB parallel_for to compute
the matrix multiplication
void ParallelMxM(float c[N][N], float a[N][N], float b[N][N] ) {
        mxm(c,a,b);
}
```


Part II, Problem 3 (see p. 59, sec 11)

```
// compute sum of data in all nodes of binary tree
void improved () {
   float sum;
      …
      tbb::task& root_task = *new (tbb::task::allocate_root ())
                                  MyRecursiveTask (tree, &sum);
      tbb::task::spawn_root_and_wait (root_task);
      …
// tbb::task* execute is a pure virtual method of tbb::task
   tbb::task* execute () { //compute x, y: partial sums for left/right sub-tree
        …
        // Task counter == number of children + 1
        int count = 1;
         if( root->left ) {
           // EXAMPLE: Allocating memory for new child to process left tree
           ++count; // Increment task counter
           // Allocate memory for the new child task and add it to the list of tasks
           // Note: the new task has the same type as the parent task.
           list.push_back (*new (allocate_child()) MyRecursiveTask (root->left, &x));
        }
        if( root->right )
           // Process the "right" tree
        // Set task counter
        set_ref_count (count);
```


Part II, Problem 5 (see Tutorial, p. 36, sec 6.1)

Code too complex …

Key ideas:

Original used parallel_for and lock

Improved is a concurrent library, no need for lock

Project 2, cont. Using OpenMP

- You can do your development on any machines, and use compilers available to you. However, the final measurements should be obtained on the quadcore systems in lab5. Here is how to invoke OpenMP for gcc and icc.
  - gcc: gcc –fopenmp prodcons.c
  - icc: icc –openmp prodcons.c

Chapter 3: Reasoning about Performance

- Recall introductory lecture:
  - Easy to write a parallel program that is slower than sequential!
- Naïvely, many people think that applying P processors to a T time computation will result in T/P time performance
- Generally wrong
  - For a few problems (Monte Carlo) it is possible to apply more processors directly to the solution
  - For most problems, using P processors requires a paradigm shift, additional code, "communication" and therefore overhead
  - Also, differences in hardware
  - Assume "P processors => T/P time" to be the best case possible
  - In some cases, can actually do better (why?)

Sources of Performance Loss

- Overhead not present in sequential computation
- Non-parallelizable computation
- Idle processors, typically due to load imbalance
- Contention for shared resources

Sources of parallel overhead

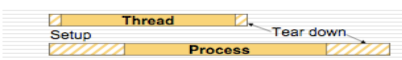
- Thread/process management (next few slides)
- Extra computation
  - Which part of the computation do I perform?
  - Select which part of the data to operate upon
  - Local computation that is later accumulated with a reduction
  - …
- Extra storage
  - Auxiliary data structures
  - "Ghost cells"
- "Communication"
  - Explicit message passing of data
  - Access to remote shared global data (in shared memory)
  - Cache flushes and coherence protocols (in shared memory)
  - Synchronization (book separates synchronization from communication)

Processes and Threads (& Filaments…)

- Let's formalize some things we have discussed before
- Threads …
  - consist of program code, a program counter, call stack, and a small amount of thread-specific data
  - share access to memory (and the file system) with other threads
  - communicate through the shared memory
- Processes …
  - Execute in their own private address space
  - Do not communicate through shared memory, but need another mechanism like message passing; shared address space another possibility
  - Logically subsume threads
  - Key issue: How is the problem divided among the processes, which includes data and work

Comparison

- Both have code, PC, call stack, local data
  - Threads -- One address space
  - Processes -- Separate address spaces
  - Filaments and similar are extremely fine-grain threads
- Weight and Agility
  - Threads: lighter weight, faster to setup, tear down, more dynamic
  - Processes: heavier weight, setup and tear down more time consuming, communication is slower



Managing Thread Overhead

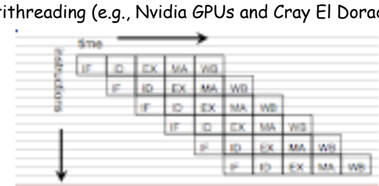
- We have casually talked about thread creation being slow and undesirable
  - So try to optimize this overhead
  - Consider static or one-time thread allocation
  - Create a pool of threads and reuse for different parallel computations
  - Works best when number of threads is fixed throughout computation

Latency vs. Throughput

- Parallelism can be used either to reduce latency or increase throughput
  - Latency refers to the amount of time it takes to complete a given unit of work (speedup).
  - Throughput refers to the amount of work that can be completed per unit time (pipelining computation).
- There is an upper limit on reducing latency
  - Speed of light, esp. for bit transmissions
  - In networks, switching time (node latency)
  - (Clock rate) x (issue width), for instructions
  - Diminishing returns (overhead) for problem instances
  - Limitations on #processors or size of memory
  - Power/energy constraints

Throughput Improvements

- Throughput improvements are often easier to achieve by adding hardware
  - More wires improve bits/second
  - Use processors to run separate jobs
  - Pipelining is a powerful technique to execute more (serial) operations in unit time
- Common way to improve throughput
  - Multithreading (e.g., Nvidia GPUs and Cray El Dorado)



Latency Hiding from Multithreading

- Reduce wait times by switching to work on different operation
  - Old idea, dating back to Multics
  - In parallel computing it's called *latency hiding*
- Idea most often used to lower λ costs
  - Have many threads ready to go …
  - Execute a thread until it makes nonlocal ref
  - Switch to next thread
  - When nonlocal ref is filled, add to ready list

Performance Loss: Contention

- Contention -- the action of one processor interferes with another processor's actions -- is an elusive quantity
  - Lock contention: One processor's lock stops other processors from referencing; they must wait
  - Bus contention: Bus wires are in use by one processor's memory reference
  - Network contention: Wires are in use by one packet, blocking other packets
  - Bank contention: Multiple processors try to access different locations on one memory chip simultaneously

Performance Loss: Load Imbalance

- Load imbalance, work not evenly assigned to the processors, underutilizes parallelism
  - The assignment of work, not data, is key
  - Static assignments, being rigid, are more prone to imbalance
  - Because dynamic assignment carries overhead, the quantum of work must be large enough to amortize the overhead
  - With flexible allocations, load balance can be solved late in the design programming cycle

Other considerations we have discussed

- Locality (next few lectures)
- Granularity of Parallelism

Summary:

- Issues in reasoning about performance