L21: Final Preparation and Course Retrospective (also very brief introduction to Map Reduce)

December 1, 2011

Administrative

- Next homework, CUDA, MPI (Ch. 3) and Apps (Ch. 6)
  - Goal is to prepare you for final
  - We'll discuss it in class on Thursday
  - Solutions due on Monday, Dec. 5 (should be straightforward if you are in class today)
- Poster dry run on Dec. 6, final presentations on Dec. 8
- Optional final report (4-6 pages) due on Dec. 15 can be used to improve your project grade if you need that

Outline

- Next homework
  - Discuss solutions
- Two problems from midterm
- Poster information
- SC Followup: Student Cluster Competition

Midterm Review, III.b

- How would you rewrite the following code to perform as many operations as possible in SIMD mode for SSE and achieve good SIMD performance? Assume the data type for all the variables is 32-bit integers, and the superword width is 128 bits. Briefly justify your solution.

```
k = ?; // k is an unknown value in the range 0 to n, and b is
       declared to be of size 2n
for (i= 0; i<n; i++) {
    x1 = i1 + j1;
    x2 = i2 + j2;
    x3 = i3 + j3;
    x4 = i4 + j4;
    x5 = x1 + x2 + x3 + x4;
    a[i] = b[n+i]*x5;
}
```

X5 is loop invariant
You must align b, and a and b likely have different alignments.

If you did not realize that X5 is loop invariant, or if you wanted to execute it in SSE SIMD mode anyway, you need to pack the "I" and "j" operands to compute the "X" values.

Midterm Review, III.c

- c. Sketch out how to rewrite the following code to improve its cache locality and locality in registers.

```
for (i=1; i<n; i++)
    for (j=1; j<n; j++)
        a[j][i] = a[j-1][i-1] + c[j];
```

- Briefly explain how your modifications have improved memory access behavior of the computation.

Interchange I and j loops:
spatial locality in a
temporal locality in c (assign to register)
Tile I loop:
temporal locality of a in j dimension
Unroll I,J loops:
Expose reuse of a in loop body, assign to registers

Homework 4, due Monday, December 5 at 11:59PM

Instructions: We'll go over these in class on December 1. Handin on CADE machines:

"handin cs4961 hw4 <probfile>"

1. Given the following sequential code, sketch out two CUDA implementations for just the kernel computation (not the host code). Determine the memory access patterns and whether you will need synchronization. Your two versions should (a) use only global memory; and, (b) use both global and shared memory. Keep in mind capacity limits for shared memory and limits on the number of threads per block.

```
…
int a[1024][1024], b[1024];
for (i=0; i<1020; i++) {
 for (j=0; j<1024-i; j++) {
        b[i+j] += a[j][i] + a[j][i+1] + a[j][i+2] + a[j][i+3];
 }
}
```

Homework 4, cont.

2. Programming Assignment 3.8, p. 148. Parallel merge sort starts with n/comm_sz keys assigned to each process. It ends with all the keys stored on process 0 in sorted order. … when a process receives another process' keys, it merges the new keys into its already sorted list of keys. … parallel mergesort … Then the processes should use tree-structured communication to merge the global list onto process 0 , which prints the result.
3. Exercise 6.27, p. 350. If there are many processes and many redistributions of work in the dynamic MPI implementation of the TSP solver, process 0 could become a bottleneck for energy returns. Explain how one could use a spanning tree of processes in which a child sends energy to its parent rather than process 0.
4. Exercise 6.30, p. 350 Determine which of the APIs is preferable for the n-body solvers and solving TSP.
   a. How much memory is required… will data fit into the memory …
   b. How much communication is required by each of the parallel algorithms (consider remote memory accesses and coherence as communication)
   c. Can the serial programs be easily parallelized by the use of OpenMP directives?Doe they need synchronization constructs such as condition variables or read-write locks?

MPI is similar

- Static and dynamic partitioning schemes
- Maintaining "best_tour" requires global synchronization, could be costly
  - May be relaxed a little to improve efficiency
  - Alternatively, some different communication constructs can be used to make this more asynchronous and less costly
  - MPI_Iprobe checks for available message rather than actually receiving
  - MPI_Bsend and other forms of send allow aggregating results of communication asynchronously

Terminated Function for a Dynamically Partitioned TSP solver with MPI (1)

```
if (My_avail_tour_count(my_stack) >= 2) {
   Fulfill_request(my_stack);
   return false;  /* Still more work */
} else { /* At most 1 available tour */
   Send_rejects();  /* Tell everyone who's requested */
                    /* work that I have none         */
   if (!Empty_stack(my_stack)) {
      return false;  /* Still more work */
   } else {  /* Empty stack */
      if (comm_sz == 1) return true;
      Out_of_work();
      work_request_sent = false;
      while (1) {
         Clear_msgs();  /* Messages unrelated to work, termination */
         if (No_work_left()) {
            return true;  /* No work left.  Quit */
```


Terminated Function for a Dynamically Partitioned TSP solver with MPI (2)

```
         } else if (!work_request_sent) {
            Send_work_request();  /* Request work from someone */
            work_request_sent = true;
         } else {
            Check_for_work(&work_request_sent, &work_avail);
            if (work_avail) {
               Receive_work(my_stack);
               return false;
            }
         }
      }  /* while */
   } /* Empty stack */
} /* At most 1 available tour */
```


Packing data into a buffer of contiguous memory

```
int MPI_Pack(
      void*          data_to_be_packed    /* in     */,
      int            to_be_packed_count   /* in     */,
      MPI_Datatype   datatype             /* in     */,
      void*          contig_buf           /* out    */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      MPI_Comm       comm                 /* in     */)
```

Unpacking data from a buffer of contiguous memory

```
int MPI_Unpack(
      void*          contig_buf           /* in     */,
      int            contig_buf_size      /* in     */,
      int*           position_p           /* in/out */,
      void*          unpacked_data        /* out    */,
      int            unpack_count         /* in     */,
      MPI_Datatype   datatype             /* in     */,
      MPI_Comm       comm                 /* in     */)
```

Course Retrospective

- Most people in the research community agree that there are at least two kinds of parallel programmers that will be important to the future of computing
  - Programmers that understand how to write software, but are naïve about parallelization and mapping to architecture (Joe programmers)
  - Programmers that are knowledgeable about parallelization, and mapping to architecture, so can achieve high performance (Stephanie programmers)
  - Intel/Microsoft say there are three kinds (Mort, Elvis and Einstein)
  - This course is about teaching you how to become Stephanie/Einstein programmers

Course Retrospective

- Why OpenMP, Pthreads, MPI and CUDA?
  - These are the languages that Einstein/Stephanie programmers use.
  - They can achieve high performance.
  - They are widely available and widely used.
  - It is no coincidence that both textbooks I've used for this course teach all of these except CUDA.

The Future of Parallel Computing

- It seems clear that for the next decade architectures will continue to get more complex, and achieving high performance will get harder.
- Programming abstractions will get a whole lot better.
  - Seem to be bifurcating along the Joe/Stephanie or Mort/Elvis/Einstein boundaries.
  - Will be very different.
- Whatever the language or architecture, some of the fundamental ideas from this class will still be foundational to the area.
  - Locality
  - Deadlock, load balance, race conditions, granularity…