Multicore Computing – CS 5966 / 6966 – Week 2 - Only Lecture (1/21/09)

`http://www.eng.utah.edu/~cs5966`

Contents

1 Class Matters

- Thanks for the interesting discussions in the Google list!
- I made a mistake in announcing email addresses containing `@cs.utah.edu`. Kindly note that the email addresses for this class are `teach-cs5966@eng.utah.edu` and `cs5966@eng.utah.edu`. If in doubt, please send email to me or the TA directly (we don't mind). Just make sure to put 5966 in the subject line.

2 Learning CILK

Today we will study the basics of programming in Cilk. Next Monday (1/26), we will study the theory behind Cilk through online material kept. **Please be sure to have read these items kept in the Week2 directory by 1/26/09**:

- `http://www.eng.utah.edu/~cs5966/LECTURES/Week2/PAPERS-AND-SLIDES/cilk-manual-5.4.6.pdf`

  Cilk 5.4.6 Reference Manual
- How to Survive the Multicore Software Revolution at `http://www.cilk.com/multicore-e-book`
- A Minicourse on Multithreaded Programming at

  `http://www.eng.utah.edu/~cs5966/LECTURES/Week2/PAPERS-AND-SLIDES/minicourse-cilk-leiserson-prokop.pdf`

3 Assignment 2 – GIVEN 1/21, DUE 1/27 – Variable Deadlines!!

The **Keys** I provide are for your Google group discussion thread subjects. I'll tell you what to post. There are also parts for individual submission.

Familiarizing with Cilk: Assignment 2 (due 1/25): Do this part, and report any troubles you may have faced - or things you observed, in the Google group. A few short sentences will do. In particular comment on what –stats tends to reveal as you increase the number from 1 to 6 (summary only).

This part requires you to have tried out existing Cilk code. The deadline is kept early so that we can identify problems you may face, early.

1. Copy over `~ganesh/5966/cilk` into your area. (This avoids many hassles I had to kludge around in order to set the library paths etc.)
2. Go into the `cilk-5.4.6` directory.
3. `touch cilksort`
4. Type `make -n cilksort` to see something like this (you can touch `getoptions.c` also if you are paranoid).

```
-------------------------------------------------------
make -n cilksort
../support/cilkclocal -DHAVE_CONFIG_H -I. -I.. -I../runtime
    -g -O2 -c -o cilksort.o cilksort.cilk
gcc -DHAVE_CONFIG_H -I. -I.. -I../runtime
    -g -O2 -MT getoptions.o -MD -MP -MF .deps/getoptions.Tpo -c -o getoptions.o getoptions.c
mv -f .deps/getoptions.Tpo .deps/getoptions.Po
rm -f cilksort
../support/cilkclocal  -g -O2   -o cilksort cilksort.o getoptions.o  -lm
-------------------------------------------------------
```

5. Having learned what `make` does, I'm just sticking in the `-cilk-profile` and `-cilk-span` options below. Type these commands

```
-------------------------------------------------------
../support/cilkclocal -cilk-span -cilk-profile  -DHAVE_CONFIG_H -I. -I.. -I../runtime
    -g -O2 -c -o cilksort.o cilksort.cilk
gcc -DHAVE_CONFIG_H -I. -I.. -I../runtime
    -g -O2 -MT getoptions.o -MD -MP -MF .deps/getoptions.Tpo -c -o getoptions.o getoptions.c
mv -f .deps/getoptions.Tpo .deps/getoptions.Po
rm -f cilksort
../support/cilkclocal  -cilk-profile -cilk-span -g -O2   -o cilksort cilksort.o getoptions.o  -lm
cilksort --stats 2 --nproc 4
cilksort --stats 2 --nproc 4 -n 3000000
cilksort --stats 2 --nproc 4 -n 30000000
cilksort --stats 3 --nproc 4 -n 30000000
-------------------------------------------------------
```

Running Measurements on Cilksort: Assignment 2 (due 1/25): Do this part, and report some of the vital statistics you obtained as a small ASCII table. Make sure that Buckley is lightly loaded. You can try Intel machines also, but the library contents has to be rebuilt. All this should work on AMD machines.

```
-------------------------------------------------------
cilksort --stats 2 --nproc 1 -n 3000000
cilksort --stats 2 --nproc 2 -n 3000000
cilksort --stats 2 --nproc 3 -n 3000000
cilksort --stats 2 --nproc 4 -n 3000000
cilksort --stats 2 --nproc 5 -n 3000000
cilksort --stats 2 --nproc 6 -n 3000000
cilksort --stats 2 --nproc 7 -n 3000000
cilksort --stats 2 --nproc 8 -n 3000000
-------------------------------------------------------
```

Running Measurements on arraysum: Assignment 2 (due 1/25): I've kept arraysum binary, but of course read protected its contents. Run these measurements and have a chat in the class group.

```
-------------------------------------------------------
arraysum --nproc 1 --stats 2 -n 10000001
arraysum --nproc 2 --stats 2 -n 10000001
etc.
-------------------------------------------------------
```

till you see diminishing returns.

Writing arraysum: Assignment 2 (due 1/27): Write a version of arraysum that splits the array recursively and adds the contents. Get ideas by studying cilksort.cilk included below. **This part requires you to mail an URL to us containing your solution, as detailed below.** This part carries **80% of the points** for this assignment, with the earlier parts carrying the rest. The 80% is split as shown below.

Do these parts. Assemble your results into a single PDF. Put the PDF at a URL. Please then email me and Sriram directly a URL for this PDF. Have your email have a subject line "5966, Writing arraysum."

1. **(10%):** Study `cilksort.cilk` and write a few paragraphs on how it is organized. You'll get ideas for your assignment from this exercise. (Feel free to modify cilksort to obtain arraysum. However, do acknowledge the sources.)
2. **(10%):** Do a binary split of the array. Add the array serially (like sequential merge) if of size below a threshold.
3. **(10%):** Do a 4-way split, keeping the same threshold for serial summation. Summarize your salient observations.
4. **(30%):** Experimentally determine, using a few experiments, where you seem to obtain the best speedup (linear) as well as reduced wall-clock time. You may vary the threshold and the splitting degree to find the sweet-spots.
5. **(20%):** For the best results in terms of speedup and runtime.
6. **(20%):** For the clearest explanations of your experimental results.

4 Anatomy of Cilksort

```
static const char *ident __attribute__((__unused__))
     = "$HeadURL: https://bradley.csail.mit.edu/svn/repos/cilk/5.4.3/examples/cilksort.cilk $ $LastChangedBy: sukhaj $ $Rev: 51
/*
 * Copyright (c) 2000 Massachusetts Institute of Technology
 * Copyright (c) 2000 Matteo Frigo
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2 of the License, or
 * (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this program; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 *
 */
/*
 * this program uses an algorithm that we call 'cilksort'.
 * The algorithm is essentially mergesort:
 *
 *   cilksort(in[1..n]) =
 *       spawn cilksort(in[1..n/2], tmp[1..n/2])
 *       spawn cilksort(in[n/2..n], tmp[n/2..n])
 *      sync
 *      spawn cilkmerge(tmp[1..n/2], tmp[n/2..n], in[1..n])
 *
 * The procedure cilkmerge does the following:
 *
 *      cilkmerge(A[1..n], B[1..m], C[1..(n+m)]) =
 *         find the median of A \union B using binary
 *         search.  The binary search gives a pair
 *         (ma, mb) such that ma + mb = (n + m)/2
 *         and all elements in A[1..ma] are smaller than
 *         B[mb..m], and all the B[1..mb] are smaller
 *         than all elements in A[ma..n].
 *
 *         spawn cilkmerge(A[1..ma], B[1..mb], C[1..(n+m)/2])
 *         spawn cilkmerge(A[ma..m], B[mb..n], C[(n+m)/2 .. (n+m)])
 *         sync
 *
 * The algorithm appears for the first time (AFAIK) in S. G. Akl and
 * N. Santoro, "Optimal Parallel Merging and Sorting Without Memory
 * Conflicts", IEEE Trans. Comp., Vol. C-36 No. 11, Nov. 1987 .  The
 * paper does not express the algorithm using recursion, but the
 * idea of finding the median is there.
 *
 * For cilksort of n elements, T_1 = O(n log n) and
 * T_\infty = O(log^3 n).  There is a way to shave a
 * log factor in the critical path (left as homework).
 */
#include <cilk-lib.cilkh>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <getoptions.h>

typedef long ELM;

/* MERGESIZE must be >= 2 */
#define KILO 1024
#define MERGESIZE (2*KILO)
#define QUICKSIZE (2*KILO)
#define INSERTIONSIZE 20

static unsigned long rand_nxt = 0;

static inline unsigned long my_rand(void)
{
     rand_nxt = rand_nxt * 1103515245 + 12345;
     return rand_nxt;
}

static inline void my_srand(unsigned long seed)
{
     rand_nxt = seed;
}

static inline ELM med3(ELM a, ELM b, ELM c)
{
     if (a < b) {
          if (b < c) {
               return b;
          } else {
               if (a < c)
                    return c;
               else
                    return a;
          }
     } else {
          if (b > c) {
               return b;
          } else {
               if (a > c)
                    return c;
               else
                    return a;
          }
     }
}
/*
 * simple approach for now; a better median-finding
 * may be preferable
 */
static inline ELM choose_pivot(ELM *low, ELM *high)
{
     return med3(*low, *high, low[(high - low) / 2]);
}

static ELM *seqpart(ELM *low, ELM *high)
{
     ELM pivot;
     ELM h, l;
     ELM *curr_low = low;
     ELM *curr_high = high;

     pivot = choose_pivot(low, high);

     while (1) {
          while ((h = *curr_high) > pivot)
               curr_high--;

          while ((l = *curr_low) < pivot)
               curr_low++;

          if (curr_low >= curr_high)
               break;

          *curr_high-- = l;
          *curr_low++ = h;
     }
     /*
      * I don't know if this is really necessary.
      * The problem is that the pivot is not always the
      * first element, and the partition may be trivial.
      * However, if the partition is trivial, then
      * *high is the largest element, whence the following
      * code.
      */
     if (curr_high < high)
          return curr_high;
     else
          return curr_high - 1;
}

#define swap(a, b) \
{ \
  ELM tmp;\
  tmp = a;\
  a = b;\
  b = tmp;\
}

static void insertion_sort(ELM *low, ELM *high)
{
     ELM *p, *q;
     ELM a, b;

     for (q = low + 1; q <= high; ++q) {
          a = q[0];
          for (p = q - 1; p >= low && (b = p[0]) > a; p--)
               p[1] = b;
          p[1] = a;
     }
}

/*
 * tail-recursive quicksort, almost unrecognizable :-)
 */
void seqquick(ELM *low, ELM *high)
{
     ELM *p;

     while (high - low >= INSERTIONSIZE) {
          p = seqpart(low, high);
          seqquick(low, p);
          low = p + 1;
     }

     insertion_sort(low, high);
}

void seqmerge(ELM *low1, ELM *high1, ELM *low2, ELM *high2,
              ELM *lowdest)
{
     ELM a1, a2;

     /*
      * The following 'if' statement is not necessary
      * for the correctness of the algorithm, and is
      * in fact subsumed by the rest of the function.
      * However, it is a few percent faster.  Here is why.
      *
      * The merging loop below has something like
      *   if (a1 < a2) {
      *        *dest++ = a1;
      *        ++low1;
      *        if (end of array) break;
      *        a1 = *low1;
      *   }
      *
      * Now, a1 is needed immediately in the next iteration
      * and there is no way to mask the latency of the load.
      * A better approach is to load a1 *before* the end-of-array
      * check; the problem is that we may be speculatively
      * loading an element out of range.  While this is
      * probably not a problem in practice, yet I don't feel
      * comfortable with an incorrect algorithm.  Therefore,
      * I use the 'fast' loop on the array (except for the last
      * element) and the 'slow' loop for the rest, saving both
      * performance and correctness.
      */
     if (low1 < high1 && low2 < high2) {
          a1 = *low1;
          a2 = *low2;
          for (;;) {
               if (a1 < a2) {
                    *lowdest++ = a1;
                    a1 = *++low1;
                    if (low1 >= high1)
                         break;
               } else {
                    *lowdest++ = a2;
                    a2 = *++low2;
                    if (low2 >= high2)
                         break;
               }
          }
     }
     if (low1 <= high1 && low2 <= high2) {
          a1 = *low1;
          a2 = *low2;
          for (;;) {
               if (a1 < a2) {
                    *lowdest++ = a1;
                    ++low1;
                    if (low1 > high1)
                         break;
                    a1 = *low1;
               } else {
                    *lowdest++ = a2;
                    ++low2;
                    if (low2 > high2)
                         break;
                    a2 = *low2;
               }
          }
     }
     if (low1 > high1) {
          memcpy(lowdest, low2, sizeof(ELM) * (high2 - low2 + 1));
     } else {
          memcpy(lowdest, low1, sizeof(ELM) * (high1 - low1 + 1));
     }
}

#define swap_indices(a, b) \
{ \
  ELM *tmp;\
  tmp = a;\
  a = b;\
  b = tmp;\
}

ELM *binsplit(ELM val, ELM *low, ELM *high)
{
     /*
      * returns index which contains greatest element <= val.  If val is
      * less than all elements, returns low-1
      */
     ELM *mid;

     while (low != high) {
          mid = low + ((high - low + 1) >> 1);
          if (val <= *mid)
               high = mid - 1;
          else
               low = mid;
     }

     if (*low > val)
          return low - 1;
     else
          return low;
}

cilk void cilkmerge(ELM *low1, ELM *high1, ELM *low2,
                    ELM *high2, ELM *lowdest)
{
     /*
      * Cilkmerge: Merges range [low1, high1] with range [low2, high2]
      * into the range [lowdest, ...]
      */

     ELM *split1, *split2;      /*
                                 * where each of the ranges are broken for
                                 * recursive merge
```

```
                               */
     long int lowsize;          /*
                                 * total size of lower halves of two
                                 * ranges - 2
                                 */

     /*
      * We want to take the middle element (indexed by split1) from the
      * larger of the two arrays.  The following code assumes that split1
      * is taken from range [low1, high1].  So if [low1, high1] is
      * actually the smaller range, we should swap it with [low2, high2]
      */

     if (high2 - low2 > high1 - low1) {
          swap_indices(low1, low2);
          swap_indices(high1, high2);
     }
     if (high1 < low1) {
          /* smaller range is empty */
          memcpy(lowdest, low2, sizeof(ELM) * (high2 - low2));
          return;
     }
     if (high2 - low2 < MERGESIZE) {
          seqmerge(low1, high1, low2, high2, lowdest);
          return;
     }
     /*
      * Basic approach: Find the middle element of one range (indexed by
      * split1). Find where this element would fit in the other range
      * (indexed by split 2). Then merge the two lower halves and the two
      * upper halves.
      */

     split1 = ((high1 - low1 + 1) / 2) + low1;
     split2 = binsplit(*split1, low2, high2);
     lowsize = split1 - low1 + split2 - low2;

     /*
      * directly put the splitting element into
      * the appropriate location
      */
     *(lowdest + lowsize + 1) = *split1;
     spawn cilkmerge(low1, split1 - 1, low2, split2, lowdest);

     spawn cilkmerge(split1 + 1, high1, split2 + 1, high2,
                     lowdest + lowsize + 2);

     sync;
     return;
}

cilk void cilksort(ELM *low, ELM *tmp, long size)
{
     /*
      * divide the input in four parts of the same size (A, B, C, D)
      * Then:
      *   1) recursively sort A, B, C, and D (in parallel)
      *   2) merge A and B into tmp1, and C and D into tmp2 (in parallel)
      *   3) merbe tmp1 and tmp2 into the original array
      */
     long quarter = size / 4;
     ELM *A, *B, *C, *D, *tmpA, *tmpB, *tmpC, *tmpD;

     if (size < QUICKSIZE) {
          /* quicksort when less than 1024 elements */
          seqquick(low, low + size - 1);
          return;
     }
     A = low;
     tmpA = tmp;
     B = A + quarter;
     tmpB = tmpA + quarter;
     C = B + quarter;
     tmpC = tmpB + quarter;
     D = C + quarter;
     tmpD = tmpC + quarter;

     spawn cilksort(A, tmpA, quarter);
     spawn cilksort(B, tmpB, quarter);
     spawn cilksort(C, tmpC, quarter);
     spawn cilksort(D, tmpD, size - 3 * quarter);
     sync;

     spawn cilkmerge(A, A + quarter - 1, B, B + quarter - 1, tmpA);
     spawn cilkmerge(C, C + quarter - 1, D, low + size - 1, tmpC);
     sync;

     spawn cilkmerge(tmpA, tmpC - 1, tmpC, tmpA + size - 1, A);
     sync;
}

void scramble_array(ELM *arr, unsigned long size)
{
     unsigned long i;
     unsigned long j;

     for (i = 0; i < size; ++i) {
          j = my_rand();
          j = j % size;
          swap(arr[i], arr[j]);
     }
}

cilk void fill_array(ELM *arr, unsigned long size)
{
     unsigned long i;

     my_srand(1);
     /* first, fill with integers 1..size */
     for (i = 0; i < size; ++i) {
          arr[i] = i;
     }

     /* then, scramble randomly */
     scramble_array(arr, size);
}

int usage(void)
{
     fprintf(stderr, "\nUsage: cilksort [<cilk-options>] [-n size] [-benchmark] [-h]\n\n");
     fprintf(stderr, "Cilksort is a parallel sorting algorithm, donned \"Multisort\", which\n");
     fprintf(stderr, "is a variant of ordinary mergesort.  Multisort begins by dividing an\n");
     fprintf(stderr, "array of elements in half and sorting each half.  It then merges the\n");
     fprintf(stderr, "two sorted halves back together, but in a divide-and-conquer approach\n");
     fprintf(stderr, "rather than the usual serial merge.\n\n");

     return -1;
}
char *specifiers[] =
{"-n", "-benchmark", "-h", 0};
int opt_types[] =
{LONGARG, BENCHMARK, BOOLARG, 0};

cilk int main(int argc, char **argv)
{
     long size;
     ELM *array, *tmp;
     long i;
     int success, benchmark, help;
     Cilk_time tm_begin, tm_elapsed;
     Cilk_time wk_begin, wk_elapsed;
     Cilk_time cp_begin, cp_elapsed;

     /* standard benchmark options */
     size = 3000000;

     get_options(argc, argv, specifiers, opt_types, &size, &benchmark, &help);

     if (help)
          return usage();

     if (benchmark) {
          switch (benchmark) {
              case 1:           /* short benchmark options -- a little work */
                   size = 10000;
                   break;
              case 2:           /* standard benchmark options */
                   size = 3000000;
                   break;
              case 3:           /* long benchmark options -- a lot of work */
                   size = 4100000;
                   break;
          }
     }
     array = (ELM *) malloc(size * sizeof(ELM));
     tmp = (ELM *) malloc(size * sizeof(ELM));

     spawn fill_array(array, size);
     sync;

     /* Timing. "Start" timers */
     sync;
     cp_begin = Cilk_user_critical_path;
     wk_begin = Cilk_user_work;
     tm_begin = Cilk_get_wall_time();

     spawn cilksort(array, tmp, size);
     sync;

     /* Timing. "Stop" timers */
     tm_elapsed = Cilk_get_wall_time() - tm_begin;
     wk_elapsed = Cilk_user_work - wk_begin;
     cp_elapsed = Cilk_user_critical_path - cp_begin;

     success = 1;
     for (i = 0; i < size; ++i)
          if (array[i] != i)
               success = 0;

     if (!success)
          printf("SORTING FAILURE");
     else {
          printf("\nCilk Example: cilksort\n");
          printf("            running on %d processor%s\n\n", Cilk_active_size, Cilk_active_size > 1 ? "s" : "");
          printf("options: number of elements = %ld\n\n", size);
          printf("Running time  = %4f s\n", Cilk_wall_time_to_sec(tm_elapsed));
          printf("Work          = %4f s\n", Cilk_time_to_sec(wk_elapsed));
          printf("Critical path = %4f s\n\n", Cilk_time_to_sec(cp_elapsed));
     }

     free(array);
     free(tmp);
     return 0;
}
```