A Cilk Response to the HPC Challenge (Class 2)

Bradley C. Kuszmaul
Massachusetts Institute of Technology
Computer Science and Artificial Intelligence Laboratory
(MIT CSAIL)
bradley@mit.edu

October 27, 2006

I implemented the 2006 HPC Challenge Class 2 benchmark [3] using the Cilk [1] multithreaded programming language. I implemented and measured all four of the Class 2 challenge benchmarks: Global HPL, Global RandomAccess, Global Stream, and Global FFT. I also implemented and measured the performance of Global PTRANS and Global DGEMM from the HPC benchmark suite [2].

Whereas last year I submitted a Cilk entry that was focused on both performance and portability, this year I focused mostly on performance. Last year I ran my programs on a wide variety of machines including a Pentium 4 with hyperthreading, a 4-processor Opteron, a 16-processor Sun machine, and a 16-processor Altix machine. This year, I benchmarked my programs on the NASA Columbia system, which consists of many Altix 3700 machines with 1.5GHz Itanium 2 processors, and 2GiByte of memory per processor. Columbia provides machines up to 512 processors, but due to lack of machine time, I used shared memory configurations of up to only 256 processors.

Table 1 shows the code size and performance for each benchmark. The code size is expressed as lines of code, including comments, C kernels and header files (`.h` files), but not including the Intel MKL or FFTW libraries.

I compiled the Cilk programs using Cilk 5.4.2.4 and gcc 3.3.3, using the "-O2" optimization flag. In some cases, I separated out serial kernels of the code into a C file, which I compiled with the Intel C compiler (icc), version 9.1, using the "-O3" optimization flag.

The problem sizes where chosen as specified in the HPC Challenge benchmark. Typically this meant that problem size was chosen to be at least $1/4$ of physical memory (that is, 0.5GiB per processor.)

The advantages of Cilk, as demonstrated by this entry are:

- Cilk is simple. Very little of the code is concerned with parallelism. Most of the code complexity is focused on making good use of cache.
- Cilk code is small.
- Cilk programs can use the best serial code for the bulk of the computational work. For example, I used the Intel MKL, FFTW, and various kernels compiled with `icc`.
- Cilk programs can express cache-efficient code, giving good performance on cost-effective machines.
- Cilk is efficient. On 32 processors, the Cilk code outperforms the MPI code for Stream, PTRANS, and RandomAccess (but Cilk falls behind on HPL). On 128 processors Cilk outperforms MPI for Stream and PTRANS, and performs comparably for RandomAccess (and again is behind for HPL.)

	HPL	DGEMM	Stream	PTRANS	RandomAccess	FFTE
Cilk lines of code	348	97	58	81	123	230
CPUS	Gflop/s	Gflop/s	GB/s	GB/s	GUPS	Gflop/s
1	4.522	5.079	0.786	0.701	0.00503	0.658
2	7.719	9.735	0.880	0.510	0.00637	0.880
4	13.645	19.699	1.796	0.919	0.01022	1.796
8	18.458	35.655	2.925	1.682	0.01625	2.925
16	16.028	64.882	4.045	3.275	0.01625	4.045
32	18.249	118.890	6.833	6.111	0.01521	6.834
64	22.371	247.988	14.020	11.612	0.02269	14.020
128		463.074	25.012	18.254	0.01080	25.888
256		942.951	44.209	27.163	0.00996	49.510
MPI-32	129.176		0.869	2.558	0.00364	4.080
MPI-128	638.902		2.158	7.523	0.01124	4.080

Figure 1: Cilk code size and performance on Altix 3700, 2 Gigabytes per processor. All benchmarks are "global" versions solving one large problem that fits into just over $1/4$th of memory. The benchmarks are listed in the order described on the HPCC web page `http://icl.cs.utk.edu/hpcc`. For a few of the larger runs, we were unable to obtain machine time before the deadline. At the bottom of the chart is shown the best MPI-based numbers from `http://icl.cs.utk.edu/hpcc/hpcc_results.cgi` for SGI Altix 3700 with 32 and 128 processors.

- Cilk scales down as well as up. Cilk achieves good performance on 1 processor.
- Cilk is portable across a wide variety of shared-memory machines.

Writing good Cilk code requires only a limited amount of expertise. I wrote most of this codes fairly over just a few days. Even the FFTE code, which is relatively complex, came together in only 2 days. The HPL code took longer: It took me a week to build a C version of the code, and it took Keith Randall a few more days to Cilkify the code. This is not too bad even compared to developing serial code, and is certainly more productive than MPI coding. Cilk enabled me to build all-new implementations of FFTE and HPL in only a few weeks of elapsed time.

Cilk's main disadvantage is that it requires a shared-memory machine. This shows up as two costs: (1) today's shared-memory machines are more expensive than today's clusters. (2) The operating system in a shared-memory machine can introduce serial bottlenecks (as shown by RandomAccess.) I believe that the technology trends favor Cilk in both of these areas. (1) Multicore processing technology will mean that in the future, every machine will be a shared memory machine at the node, and it seems likely that hardware makers provide ways to build shared memory clusters as cheaply as distributed-memory clusters. (2) The operating systems will improve as shared-memory machines become more widespread. In the future, the costs will drop for shared memory.

The rest of this paper examines each benchmark. The code itself appears in the appendices of this paper.

1 HPL

The code for performing LU decomposition with pivoting appears in Appendix A. The code consists of a cilk file, a C file, and a header file. Our code employs the Intel Math Kernel Library (MKL) for the base case of its matrix-multiplication function.

This code, which is new this year, on is based on an algorithm developed by Sivan Toledo and published in [4]. The Toledo algorithm is numerically stable, since its algorithm is simply a rescheduling of the same data flow graph from the naive LU decomposition. The algorithm makes efficient use of cache, and exhibits enough parallelism to keep processors busy.[1]

The code was developed as follows: I implemented a C version of the Toledo algorithm. Keith Randall (who was my colleague in graduate school and was one of the developers for Cilk 5) corrected, Cilkified and optimized the code.

HPL is the most interesting benchmark presented here. The algorithm is interesting, and would be difficult to implement using any kind of static load balancing. This algorithm shows one of strengths of Cilk: sophisticated algorithms can be expressed.

To avoid cache associativity problems, I chose the matrix side length N to be the smallest number that satisfies the following constraints:

- Use at least $1/4$ of memory: That is $N \geq \sqrt{P2^{29}/8}$, where P is the number of processors.
- Stay at least 100 away from powers of two. That is $|N - 2^i| > 100$ for all i.
- N is a prime number.

The code we used for the final submission exhibits good speedup up to about 8 processors. Cilk's critical-path-length instrumentation indicates that our code has enough parallelism, and so perhaps the problem is that the Altix is exhibiting some scaling problems.

2 DGEMM

My code for DGEMM appears in Appendix B. The code employs a divide-and-conquer approach, and is very similar to DGEMM code that appears in our HPL code. The base case of the recursion is an $m \times k \times n$ DGEMM where $m + k + n < 512$. My code employs the MKL to implement the base case of the recursion.

My code achieves speedup of 92 on 128 processors.

3 Stream

The code for Stream appears in Appendix C. Stream employs no other libraries.

Stream achieves poor speedup when transitioning from 1 to 2 processors, probably because the two processors share a memory bus. Then Stream achieves fairly linear speedups until the number of processors is large. As described below in Section 5, I suspect that for large numbers of processors the operating system's TLB miss handler is limiting the speedup.

4 PTRANS

The code for PTRANS appears in Appendix D. PTRANS employs no other libraries.

The trick for achieving good performance on PTRANS is to avoid cache associativity conflicts. To do so, I stop the divide-and-conquer recursion near at about the size that fits in cache, and copy the data to an array with short stride, transpose it, and copy it back.

[1] Last year I submitted Svetoslav Tzvetkov's code, which exhibits less parallelism, and makes less efficient use of cache.

5 RandomAccess

The code for randomaccess appears in Appendix E. RandomAccess employs no other libraries.

The performance of my randomaccess implementation does not scale with the number of processors. The randomaccess code achieves a maximum performance of 0.022 GUPS on 64 processors, and then gets worse for larger numbers of processors.

I suspect that the performance is limited by the rate at which the operating system can handle TLB misses. One possible explanation is that the operating system serializes TLB misses. Under this theory, the Linux 2.6.5 ia64 kernel can only process only one TLB miss at a time, and it takes about 50μs to handle a TLB miss. If true, then for large numbers of processors, nearly half the performance of the STREAM benchmark goes into handling serialized TLB misses as well. This performance problem seems to be an operating system issue, not a hardware or programming language issue.

6 FFTE

The code for FFTE appears in Appendix F. FFTE employs FFTW, which provides a Cilk interface to achieve parallelism when performing an many FFTs on an array.

One problem I faced for FFTE is that the FFTW program is not optimized for large one-dimensional FFT problems. So I implemented the 4-step solution: Compute FFTs on the rows. Transpose and multiply by the twiddle factors. Compute FFTs on the columns.

Since serial FFT is most efficient on arrays which are a power of two, it was especially important to get the transpose right, using the technique mentioned above for PTRANS. Otherwise, cache associativity misses would hurt the performance.

7 Acknowledgments

Keith Randall contributed substantially to the LU-decomposition implementation.

Chris Hill at MIT's department of Earth and Planetary Sciences helped run the benchmarks on the NASA Columbia machine.

I used FFTW and the Intel MKL for the serial parts of certain benchmarks.

References

[1] The Cilk Project. `http://supertech.csail.mit.edu/cilk`, 2006.

[2] HPC Challenge Benchmark. `http://icl.cs.utk.edu/hpcc/`, 2006.

[3] HPC Challenge Award Competition. `http://www.hpcchallenge.org/`, 2006.

[4] Sivan Toledo. Locality of reference in LU decomposition with partial pivoting. *SIAM J. Matrix Analysis and Applications*, 18(4):1065–1081, October 1997.

A HPL code

The HPL code consists of three files, `palu.cilk`, `palu-kernels.c`, and `palu-kernels.h`.

A.1 palu.cilk

```
1  /* Copyright 2006 Keith Randall and Bradley C. Kuszmaul. */
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <math.h>
5  #include <string.h>
6  #include <sys/time.h>
7  #include <assert.h>
8
9  #include <cilk-lib.cilkh>
10
11 #define BASE 64
12
13 #include "palu-kernels.h"
14
15 cilk void lu(REAL *A, int *P, int n, int m, long rowsep);
16 cilk void mulsub(REAL *A, REAL *B, REAL *C, int n, int k, int m, long rowsep);
17 cilk void backsolve(REAL *A, REAL *L, REAL *M, int n, int m, long rowsep);
18 cilk void permute(REAL *A, int n, int m, int *P, long rowsep);
19
20 // assumes n >= m
21 // A and L are n*m.  U is m*m.  P is n.
22 cilk void lu(REAL *A, int *P, int n, int m, long rowsep) {
23  REAL *AORIG;
24  if (TEST) {
25    AORIG = malloc(n * rowsep * sizeof(*AORIG));
26    memcpy(AORIG, A, n * rowsep * sizeof(*AORIG));
27  }
28  if (m == 1 || (long long)n * m * m <= BASE*BASE*BASE) {
29    lu_ser(A, P, n, m, rowsep);
30  } else {
31    int i;
32    int k = m >> 1;
33
34    REAL *A_00 = &A_(0,0);      // k x k
35    REAL *A_10 = &A_(k,0);      // n-k x k
36    REAL *A_01 = &A_(0,k);      // k x m-k
37    REAL *A_11 = &A_(k,k);      // n-k x m-k
38    Cilk_time cp = Cilk_user_critical_path;
39    int *Q = Cilk_alloca(n * sizeof(*Q));
40    int *R = Cilk_alloca((n - k) * sizeof(*Q));
41    sync;
42
43    // compute L_00, L_10, and U_00 from A_00 and A_10.  Also
44    // permute A_00, A_10, and generate P[0..n).
45    spawn lu(A_00, Q, n, k, rowsep); sync;
46
47    // permute A_01 and A_11
48    spawn permute(A_01, n, m - k, Q, rowsep); sync;
49
50    // backsolve for U_01 in A_01 = L_00 U_01
51    spawn backsolve(A_01, A_00, A_01, k, m - k, rowsep); sync;
52
```

```
53      // compute A_11 -= L_10 U_01
54      spawn mulsub(A_10, A_01, A_11, n - k, k, m - k, rowsep); sync;
55
56      // compute L_11 and U_11 by recursing on A_11 - L_10 U_01 = L_11 U_11
57      spawn lu(A_11, R, n - k, m - k, rowsep); sync;
58
59      // permute L_10
60      spawn permute(A_10, n - k, k, R, rowsep); sync;
61
62      // combine permutations: Q first, then R (P = RQ)
63      for (i = 0; i < n; i++) {
64        int j = Q[i];
65        if (j >= k) {
66          j = R[j - k] + k;
67        }
68        P[i] = j;
69      }
70    }
71  }
72
73  // Given A and L, solve for M in A=LM.  L is lower triangular, 1 on diagonal.
74  // A is n*m
75  // L is n*n
76  // M is n*m
77  cilk void backsolve(REAL *A, REAL *L, REAL *M, int n, int m, long rowsep) {
78    REAL *AORIG;
79    if (TEST) {
80      AORIG = malloc(n * rowsep * sizeof(*AORIG));
81      memcpy(AORIG, A, n*rowsep*sizeof(*AORIG));
82    }
83    if ((long long)n * n * m <= BASE*BASE*BASE) {  // base case
84      backsolve_ser(A, L, M, n, m, rowsep);
85    } else {
86      int k = m >> 1;
87      REAL *A_00 = &A_(0, 0);    // k x k
88      REAL *A_10 = &A_(k, 0);    // n-k x k
89      REAL *A_01 = &A_(0, k);    // k x m-k
90      REAL *A_11 = &A_(k, k);    // n-k x m-k
91      REAL *L_00 = &L_(0, 0);    // k x k
92      REAL *L_10 = &L_(k, 0);    // n-k x k
93      REAL *L_11 = &L_(k, k);    // n-k x n-k
94      REAL *M_00 = &M_(0, 0);    // k x k
95      REAL *M_01 = &M_(0, k);    // k x m-k
96      REAL *M_10 = &M_(k, 0);    // n-k x k
97      REAL *M_11 = &M_(k, k);    // n-k x m-k
98
99      spawn backsolve(A_00, L_00, M_00, k, k, rowsep);
100     spawn backsolve(A_01, L_00, M_01, k, m - k, rowsep);
101     sync;
102     spawn mulsub(L_10, M_00, A_10, n - k, k, k, rowsep);
103     spawn mulsub(L_10, M_01, A_11, n - k, k, m - k, rowsep);
104     sync;
105     spawn backsolve(A_10, L_11, M_10, n - k, k, rowsep);
106     spawn backsolve(A_11, L_11, M_11, n - k, m - k, rowsep);
107     sync;
108   }
109   if (TEST) { // Compute LM and compare to AORIG
110     int i,j,k;
```

```
    for (i = 0; i < n; i++) {
      for (j = 0; j < m; j++) {
        REAL z = 0.0;
        for (k = 0; k < i; k++) {
          z += L_(i,k) * M_(k, j);
        }
        z += M_(i, j);
        if (fabs(z - AORIG_(i,j)) > 1e-5) {
          exit(1);
        }
      }
    }
    free(AORIG);
  }
}

// This lock table is used to avoid conflicts between to threads both trying
// to do -= on the same matrix chunk at the same time.
#define LOCK_HASH_SIZE 10007
Cilk_lockvar lock_hashtable[LOCK_HASH_SIZE];

// compute C -= AB.
// A is n*k
// B is k*m
// C is n*m

cilk void mulsub(REAL *A, REAL *B, REAL *C, int n, int k, int m, long rowsep) {
  if ((long long)n * k * m <= BASE*BASE*BASE) { // base case
    Cilk_lock(lock_hashtable[((unsigned long)C) % LOCK_HASH_SIZE]);
    mulsub_ser(A, B, C, n, k, m, rowsep);
    Cilk_unlock(lock_hashtable[((unsigned long)C) % LOCK_HASH_SIZE]);
  } else if (n >= k && n >= m) {
    int n2 = n >> 1;
    spawn mulsub(A, B, C, n2, k, m, rowsep);
    spawn mulsub(&A_(n2, 0), B, &C_(n2, 0), n - n2, k, m, rowsep);
    sync;
  } else if (m >= k) {
    int m2 = m >> 1;
    spawn mulsub(A, B, C, n, k, m2, rowsep);
    spawn mulsub(A, &B_(0, m2), &C_(0, m2), n, k, m - m2, rowsep);
    sync;
  } else {
    int k2 = k >> 1;
    spawn mulsub(A, B, C, n, k2, m, rowsep);
    //sync; // handled with locks around the basecase above.
    spawn mulsub(&A_(0, k2), &B_(k2, 0), C, n, k - k2, m, rowsep);
    sync;
  }
}

cilk void permute(REAL *A, int n, int m, int *P, long rowsep) {
  if (m <= BASE) {
    permute_ser(A, n, m, P, rowsep);
  } else {
    int m2 = m >> 1;
    spawn permute(A, n, m2, P, rowsep);
    spawn permute(&A_(0, m2), n, m - m2, P, rowsep);
    sync;
```

```
 }
}

cilk int main(int argc, char *argv[]) {
 int n = atoi(argv[1]);
 int m = atoi(argv[2]);
 long rowsep = m;
 printf("PALU factoring %d x %d\n", n, m);
 {
   int i;
   for (i = 0; i < LOCK_HASH_SIZE; i++) {
     Cilk_lock_init(lock_hashtable[i]);
   }
 }
 {
   REAL *A = malloc(n * (long)m * sizeof(*A));
   int *P = malloc(n * sizeof(*P));
   REAL *AORIG=0;
   int i, j;
   Cilk_time tm_begin, tm_elapsed;
   Cilk_time wk_begin, wk_elapsed;
   Cilk_time cp_begin, cp_elapsed;

   assert(A);
   for (i = 0; i < n; i++) {
     for (j = 0; j < m; j++) {
       A_(i, j) = ((double)random()) / RAND_MAX;
     }
   }

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

   spawn lu(A, P, n, m, rowsep); 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;

   printf("PALU(%d,%d) on %d processors\n", n, m, Cilk_active_size);
   printf("Running time  = %4f ms\n", 1000*Cilk_wall_time_to_sec(tm_elapsed));
   printf("Work          = %4f ms\n", 1000*Cilk_time_to_sec(wk_elapsed));
   printf("Critical path = %4f ms\n", 1000*Cilk_time_to_sec(cp_elapsed));
   printf("Parallelism   = %4f\n",    (double)wk_elapsed/(double)cp_elapsed);
   printf("gflops = %g\n", (2.0e-9*n*m*m)/3.0 / Cilk_wall_time_to_sec(tm_elapsed));
 }
 return 0;
}
```

A.2 palu-kernels.c

```
/* Copyright 2006 Keith Randall and Bradley C. Kuszmaul. */
#include <mkl.h>
#include "palu-kernels.h"
```

```
4
5   static inline REAL rabs(REAL x) {
6      return (x >= 0) ? x : -x;
7   }
8
9   // O(nzm)
10  void mulsub_ser(REAL *A, REAL *B, REAL *C, int n, int z, int m, long rowsep) {
11   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans, n, m, z, -1.0, A, rowsep, B, rowsep, 1.0, C, rowsep);
12  }
13
14  // O(n^2m)
15  void backsolve_ser(REAL *A, REAL *L, REAL *M, int n, int m, long rowsep) {
16   int i, j, k;
17   for (i = 0; i < n; i++) {
18     for (j = 0; j < m; j++) {
19       double r = 0.0;
20       for (k = 0; k < i; k++) {
21         r += L_(i, k) * M_(k, j);
22       }
23       M_(i, j) = A_(i, j) - r;
24     }
25   }
26  }
27
28  // This version is slightly faster, where we postpone the last big
29  // matrix update until we actually need the data for that row/column.
30  // At that point  we "materialize" all the data we need, doing the
31  // computations we skipped earlier.
32  void lu_ser(REAL *A, int *P, int n, int m, long rowsep) {
33   int i, j, k;
34   for (i = 0; i < n; i++) P[i] = i; // initialize P
35   for (i = 0; i < m; i++) {
36
37     // materialize column i (from row i to row n).  Need to materialize
38     // in increasing order, as subsequent materializations need older data.
39     for (j = i; j < n; j++) {
40       REAL z = 0;
41       for (k = 0; k < i; k++) {
42         z += A_(k, i) * A_(j, k);
43       }
44       A_(j, i) -= z;
45     }
46
47     // find pivot row (largest value in column i from row i to n)
48     REAL p = rabs(A_(i, i));
49     int prow = i;
50     for (j = i + 1; j < n; j++) {
51       REAL q = rabs(A_(j, i));
52       if (q > p) {
53         p = q;
54         prow = j;
55       }
56     }
57
58     // swap rows i and prow (if needed)
59     if (i != prow) {
60       for (k = 0; k < m; k++) {
61         REAL t = A_(i, k);
```

```
        A_(i, k) = A_(prow, k);
        A_(prow, k) = t;
      }
      // record swap in permutation array
      // TODO: faster way to do this?
      for (k = 0; k < n; k++) {
        if (P[k] == i) P[k] = prow;
        else if (P[k] == prow) P[k] = i;
      }
    }

    // materialize the rest of pivot row
    for (j = i + 1; j < m; j++) {
      REAL z = 0;
      for (k = 0; k < i; k++) {
        z += A_(k, j) * A_(i, k);
      }
      A_(i, j) -= z;
    }

    // divide out pivot
    // TODO: handle p==0 case?
    p = 1 / A_(i, i);
    for (j = i + 1; j < n; j++) {
      A_(j, i) *= p;
    }
  }
}

// A permutation P is interpreted as follows.  P[i] = j means
// that row i of A should be equal to row j of PA.  Read P as
// "P[i] == j then row i goes to row j".
void permute_ser(REAL *A, int n, int m, int *P, long rowsep) {
  REAL *tmp = alloca(m * sizeof(*tmp));
  char *z = alloca(n * sizeof(*z));
  int i;

  bzero(z, n * sizeof(*z));
  for (i = 0; i < n; i++) {
    int desti;
    if (z[i] != 0) continue;  // already processed in an earlier chain
    desti = P[i];
    while (desti != i) {
      z[desti] = 1;
      memcpy(tmp, &A_(desti, 0), m*sizeof(*A));       // save row desti
      memcpy(&A_(desti, 0), &A_(i, 0), m*sizeof(*A)); // move i -> desti
      memcpy(&A_(i, 0), tmp, m*sizeof(*A));           // keep old desti in i
      desti = P[desti];
    }
  }
}
```

A.3 palu-kernels.h

```
/* Copyright 2006 Keith Randall and Bradley C. Kuszmaul. */
typedef double REAL;

#define A_(i,j) (A[(i)*(long)rowsep + (j)])
```

```
#define B_(i,j) (B[(i)*(long)rowsep + (j)])
#define C_(i,j) (C[(i)*(long)rowsep + (j)])
#define AORIG_(i,j) (AORIG[(i)*(long)rowsep + (j)])
#define CORIG_(i,j) (CORIG[(i)*(long)rowsep + (j)])
#define LU_(i,j) (LU[(i)*(long)rowsep + (j)])
#define L_(i,j) (L[(i)*(long)rowsep + (j)])
#define M_(i,j) (M[(i)*(long)rowsep + (j)])

extern void mulsub_ser(REAL *A, REAL *B, REAL *C, int n, int z, int m, long rowsep);
extern void backsolve_ser(REAL *A, REAL *L, REAL *M, int n, int m, long rowsep);
extern void lu_ser(REAL *A, int *P, int n, int m, long rowsep);
extern void permute_ser(REAL *A, int n, int m, int *P, long rowsep);
```

B dgemm.cilk

```
/* Copyright (c) 2006 Bradley C. Kuszmaul */
#include <mkl.h>
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>

CBLAS_ORDER order = CblasColMajor;
CBLAS_TRANSPOSE transA = CblasNoTrans, transB = CblasNoTrans;
double beta=1;

#define BASE 512

cilk void mminto (double *A, double *B, double *C, int m, int n, int k, double alpha, long columnsep)
// C += alpha*(A*B)
//  A is m by k
//  B is k by n
//  C is m by n
{
   if (m+n+k<BASE) {
       cblas_dgemm(order, transA, transB, m, n, k, alpha,
                   A, columnsep, B, columnsep, beta, C, columnsep);
   } else if (m>=n && m>=k) {
       /* The biggest dimension is m. */
       spawn mminto(A,     B, C,     m/2,   n, k, alpha, columnsep);
       spawn mminto(A+m/2, B, C+m/2, m-m/2, n, k, alpha, columnsep);
   } else if (n>=m && n>=k) {
       /* The biggest dimension is n */
       spawn mminto(A, B,                 C,                 m, n/2,   k, alpha, columnsep);
       spawn mminto(A, B+(n/2)*columnsep, C+(n/2)*columnsep, m, n-n/2, k, alpha, columnsep);
   } else {
       /* The biggest dimension is k. */
       spawn mminto(A                , B,     C, m, n, k/2,   alpha, columnsep);
       // Need to store into another variable then add them.  Or a sync.
       sync;
       spawn mminto(A+(k/2)*columnsep, B+k/2, C, m, n, k-k/2, alpha, columnsep);
   }
}

int difflarge(double a, double b) {
   double d = a-b;
```

```
43      double relda = d/a;
44      double reldb = d/b;
45      if (d<0) d=-d;
46      if (relda<0) relda=-relda;
47      if (reldb<0) reldb=-reldb;
48      if (d<1e-9) return 0;
49      if (relda>1e-9) return 1;
50      if (reldb>1e-9) return 1;
51      return 0;
52  }
53
54  double *A, *B, *Cme;
55
56  cilk void init1 (long i, long n) {
57   long j;
58   for (j=0; j<n; j++) {
59     A[i+j] = 3*(i+j);//(double)(random())/RAND_MAX;
60     B[i+j] = 3*(i+j)+1;//(double)(random())/RAND_MAX;
61     Cme[i+j] = 3*(i+j)+2;//(double)(random())/RAND_MAX;
62   }
63  }
64
65  cilk void init (long n) {
66   long i;
67   for (i=0; i<n*n; i+=n) {
68     spawn init1(i, n);
69   }
70  }
71
72  cilk void test_n(int n) {
73     double alpha = -1;
74     int i;
75     struct timeval tv0,tv1,tv2;
76     double tdiff_rec, tdiff_mkl;
77     long long n3 = ((long long)n)*((long long)n)*((long long)n);
78     A=malloc((long)n*(long)n*sizeof(double));
79     B=malloc((long)n*(long)n*sizeof(double));
80     Cme=malloc((long)n*(long)n*sizeof(double));
81     assert(A && B && Cme);
82     spawn init(n); sync;
83     gettimeofday(&tv0,0);
84     spawn mminto(A, B, Cme, n, n, n, alpha, n);
85     sync;
86     gettimeofday(&tv1,0);
87     tdiff_rec = tv1.tv_sec-tv0.tv_sec + 1e-6*(tv1.tv_usec-tv0.tv_usec);
88     printf("%dx%d matrix multiply %lld flops in %fs = %fMFLOPS\n",
89            n,n, n3, tdiff_rec, 2*n3*1e-6/tdiff_rec);
90  }
91
92  cilk int main (int argc, char *argv[]) {
93     assert(argc==2);
94     spawn test_n(atoi(argv[1]));
95     sync;
96     return 0;
97  }
```

C stream.cilk

```
1   /* Copyright (c) 2006 Bradley C. Kuszmaul */
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include <cilk-lib.cilkh>
5   #include <assert.h>
6
7   double alpha;
8   #define BASE 128
9   cilk void doit (double *a, double *b, double *c, long n) {
10      if (n<BASE) {
11          long i;
12          for (i=0; i<n; i++) {
13              a[i]=b[i]+alpha*c[i];
14          }
15      } else {
16          spawn doit (a, b, c, n/2);
17          spawn doit (a+n/2, b+n/2, c+n/2, n-n/2);
18      }
19  }
20
21  int n_trials;
22  long n_elements;
23  double *A,*B,*C;
24
25  void init (void) {
26      long i;
27      A = malloc (sizeof(*A)*n_elements);
28      B = malloc (sizeof(*B)*n_elements);
29      C = malloc (sizeof(*C)*n_elements);
30      assert(A!=0); assert(B!=0); assert(C!=0);
31      for (i=0; i<n_elements; i++) { B[i] = drand48();    C[i] = drand48(); }
32  }
33
34  cilk void run_trials (void) {
35      int i;
36      Cilk_time start_t, end_t;
37      double tdiff;
38      printf("n_elements=%ld\n", n_elements);
39      init();
40      spawn doit(A,B,C,n_elements); sync;
41
42      start_t = Cilk_get_wall_time();
43      for (i=0; i<n_trials; i++) {
44          spawn doit(A,B,C,n_elements);
45          sync;
46      }
47      end_t = Cilk_get_wall_time();
48      tdiff = Cilk_wall_time_to_sec(end_t - start_t);
49      printf("%ld elements %ldGB %.9fs %d trials, %.9fsGB/s\n",
50             n_elements, 24L*n_elements, tdiff, n_trials, ((24e-9*n_trials)*n_elements)/tdiff);
51  }
52
53  cilk int main (int argc, char *argv[]) {
54      n_elements = strtol(argv[1], 0, 10);
55      spawn run_trials();
56      sync;
```

```
57      return 0;
58  }
```

D PTRANS code

The PTRANS code consists of three files, `ptrans.cilk`, `ptrans-base.c`, and `ptrans-base.h`.

D.1 ptrans.cilk

```
1   /* Copyright (c) 2006 Bradley C. Kuszmaul */
2   #include <stdlib.h>
3   #include <stdio.h>
4   #include "ptrans-base.h"
5
6   typedef double val_t;
7
8   // The trick for performance is to stop at near the level-1 cache size, and copy
9   // the stuff into a buffer of the right size.
10
11  // Compute A += tranpose(B)
12  cilk void ptrans (val_t *A, const val_t *B, int rowlen, int arows, int brows) {
13   if ((arows+brows)<BASE) {
14     ptrans_base(A,B,rowlen,arows,brows);
15   } else {
16     int a2 = arows/2;
17     int b2 = brows/2;
18     spawn ptrans (A,                   B,                     rowlen, a2,       b2);
19     spawn ptrans (A+a2,                B+   a2*(long)rowlen, rowlen, arows-a2, b2);
20     spawn ptrans (A+  +b2*(long)rowlen, B+b2,                 rowlen, a2,       brows-b2);
21     spawn ptrans (A+a2+b2*(long)rowlen, B+b2+a2*(long)rowlen, rowlen, arows-a2, brows-b2);
22     sync;
23   }
24  }
25
26  cilk void init (double *A, double *B, long i, long N) {
27   if (N==1) {
28     *A = i;
29     *B = i*1e6;
30   } else {
31     spawn init (A, B, i, N/2);
32     spawn init (A+N/2, B+N/2, i+N/2, N-N/2);
33   }
34  }
35
36  cilk int main (int argc, char *argv[]) {
37     long N;
38     double *A,*B;
39     int i;
40     Cilk_time tm_before, tm_begin, tm_end;
41     N = strtol(argv[1], 0, 0);
42     printf("   N=%ld\n", N);
43     //#endif
44     A = malloc(sizeof(*A)*N*N);
45     B = malloc(sizeof(*A)*N*N);
46     if (A==0 || B==0) { printf("Couldn't allocate memory (probably too much memory)\n"); exit(1); }
47     printf("A=%p B=%p\n", A, B);
48     tm_before = Cilk_get_wall_time();
```

```
49      spawn init(A,B,0,N*N); sync;
50      tm_begin = Cilk_get_wall_time();
51      printf("init took %fs\n", Cilk_wall_time_to_sec(tm_begin-tm_before));
52      printf("   %fGiB memory\n", ((double)N)*((double)N)*2.0*sizeof(*A)/(1<<30));
53      for (i=0; i<4; i++) {
54        tm_begin = Cilk_get_wall_time();
55        spawn ptrans(A,B,N,N,N);
56        sync;
57        tm_end = Cilk_get_wall_time();
58        printf("   Time = %f", Cilk_wall_time_to_sec(tm_end-tm_begin));
59        printf("   Bandwidth=%lfGB/s\n", ((double)N)*((double)N)*sizeof(*A)*1e-9/Cilk_wall_time_to_sec(tm_end-tm_begin)
60      }
61      return 0;
62    }
```

D.2 ptrans-base.c

```
1   /* Copyright (c) 2006 Bradley C. Kuszmaul */
2   #include "ptrans-base.h"
3
4   void ptrans_base (double *A, const double *B, int rowlen, int arows, int brows) {
5    double tmp[BASE][BASE];
6     int i,j;
7     for (j=0; j<arows; j++) {
8       for (i=0; i<brows; i++) {
9         tmp[i][j] = B[i+rowlen*j];
10      }
11    }
12    for (i=0; i<arows; i++) {
13      for (j=0; j<brows; j++) {
14        A[j+i*rowlen] += tmp[i][j];
15      }
16    }
17  }
```

D.3 ptrans-base.h

```
1   #define BASE 32
2   extern void ptrans_base(double *,const double *,int, int,int);
```

E randomaccess.cilk

```
1   /* Copyright (c) 2006 Bradley C. Kuszmaul */
2   #include <cilk-lib.cilkh>
3   #include <stdlib.h>
4   #include <stdio.h>
5
6   typedef unsigned long long u64Int;
7   typedef signed   long long s64Int;
8
9   #define POLY 0x0000000000000007ULL
10  #define PERIOD 1317624576693539401LL
11
12  /* Utility routine to start random number generator at Nth step */
13  /* Analysis:  The loops of this routine run at most 64 iterations. */
14  u64Int HPCC_starts(s64Int n)
15  {
```

```
16    int i, j;
17    u64Int m2[64];
18    u64Int temp, ran;
19
20    while (n < 0) n += PERIOD;
21    while (n > PERIOD) n -= PERIOD;
22    if (n == 0) return 0x1;
23
24    temp = 0x1;
25    for (i=0; i<64; i++) {
26      m2[i] = temp;
27      temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
28      temp = (temp << 1) ^ ((s64Int) temp < 0 ? POLY : 0);
29    }
30
31    for (i=62; i>=0; i--)
32      if ((n >> i) & 1)
33        break;
34
35    ran = 0x2;
36    while (i > 0) {
37      temp = 0;
38      for (j=0; j<64; j++)
39        if ((ran >> j) & 1)
40          temp ^= m2[j];
41      ran = temp;
42      i -= 1;
43      if ((n >> i) & 1)
44        ran = (ran << 1) ^ ((s64Int) ran < 0 ? POLY : 0);
45    }
46
47    return ran;
48  }
49
50  #define NUPDATE (4 * TableSize)
51
52  /* Perform updates to the main table using divide-and-conquer.
53   * HPCC_starts() initializes at an arbitrary starting point.
54   * BASE_SIZE is the size of the base case for the recursion. */
55  #define BASE_SIZE (1<<24)
56
57  cilk void RandomAccessRecur(u64Int TableSize, u64Int *Table, u64Int low, u64Int hi) {
58      if (low + BASE_SIZE >= hi) {
59          u64Int ran = HPCC_starts(low);
60          //u64Int ranold = ran;
61          u64Int i;
62          for (i=low; i<hi; i++) {
63              // Hand optimized (Some compilers, such as gcc 3.3.3 manage to compile the
64              // conditional branch out of the code.  Use the hand-optimized code anyway.)
65              ran    = (ran << 1) ^ ((((s64Int) ran)>>63) & POLY);
66              //ranold = (ranold << 1) ^ (((s64Int) ranold < 0) ? POLY : 0);
67              //assert(ran==ranold);
68              Table[ran & (TableSize-1)] ^= ran;
69          }
70      } else {
71          spawn RandomAccessRecur(TableSize, Table, low, (low+hi)/2);
72          spawn RandomAccessRecur(TableSize, Table, (low+hi)/2, hi);
73          sync;
```

```
74      }
75  }
76
77  cilk void Init (u64Int TableSize, u64Int *Table) {
78     if (TableSize<(1<<15)) {
79         int i;
80         for (i=0; i<TableSize; i++) Table[i] = i;
81     } else {
82         u64Int t2 = TableSize/2;
83         spawn Init(t2, Table);
84         spawn Init(TableSize-t2, Table+t2);
85     }
86  }
87
88  cilk void RandomAccessUpdate (u64Int TableSize, u64Int *Table) {
89      u64Int i;
90
91      /* Initialize main table */
92      for (i=0; i<TableSize; i++) Table[i] = i;
93
94      /* Perform updates to the main table. */
95      spawn RandomAccessRecur(TableSize, Table, 0, NUPDATE);
96
97      sync;
98  }
99
100 cilk int main (int argc, char *argv[]) {
101     u64Int TableSize;
102     u64Int *t1;
103
104     double cilk_time_in_secs;
105
106     TableSize = strtol(argv[1], 0, 10);
107     t1 = malloc(sizeof(*t1)*TableSize);
108     {
109         Cilk_time start_t, end_t;
110         start_t = Cilk_get_wall_time();
111         spawn RandomAccessUpdate(TableSize, t1);
112         sync;
113         end_t  = Cilk_get_wall_time();
114         cilk_time_in_secs = Cilk_wall_time_to_sec(end_t-start_t);
115         printf("   running on %d processor%s\n",
116                Cilk_active_size, Cilk_active_size > 1 ? "s" : "");
117         printf("   time = %fs\n", cilk_time_in_secs);
118         printf("   Cilk GUPS=%f\n", (1e-9*NUPDATE)/cilk_time_in_secs);
119     }
120
121     printf("\n");
122     return 0;
123 }
```

F FFTE code

The FFTE code consists of three files, `ffte.cilk`, `transpose-base.c`, and `transpose-base.h`

F.1 fft.cilk

```
1   /* Copyright (c) 2006 Bradley C. Kuszmaul */
2   #include <cilk-lib.cilkh>
3   #include <stdio.h>
4   #include <math.h>
5   #include <assert.h>
6   #include <errno.h>
7   #include <string.h>
8   #include <fftw_cilk.cilkh>
9   #include "transpose-base.h"
10
11  // The trick for performance is to stop at near the level-1 cache size, and copy
12  // the stuff into a buffer of the right size.
13  // On Itanium, I don't know what that means, since maybe memcpy avoids L1.
14  // L2 is supposed to be 256KB
15  // If I go down to 64KB subpiece which is (1<<16)/sizeof(val_t)   (Perhaps 8 bytes)
16  //  which is 1<<13 elements.  Take the square root which is 1<<6 elements on a side
17
18  cilk void trans_offdiag (val_t *A, val_t *B, long N/*rowlen*/, long A_nrows, long B_nrows, long depth) {
19     /* A_nrows is the number of rows in A and the number of columns in B. */
20     /* Also, generally B_nrows <= A_nrows <= B_nrows+1 */
21     assert(B_nrows <= A_nrows); assert(A_nrows <= B_nrows+1);
22     if (A_nrows<BASE) {
23         tran_base(A,B,N,A_nrows,B_nrows);
24     } else {
25         long A2 = A_nrows/2;
26         long B2 = B_nrows/2;
27         spawn trans_offdiag(A,           B,           N, A2,         B2,         depth+1);
28         spawn trans_offdiag(&A[A2+0*N],  &B[0 +A2*N], N, A_nrows-A2, B2,         depth+1);
29         // swap order to maintain  B_nrows <= A_nrows <= B_nrows+1
30         spawn trans_offdiag(&B[B2+ 0*N], &A[0 +B2*N], N, B_nrows-B2, A2,         depth+1);
31         spawn trans_offdiag(&A[A2+B2*N], &B[B2+A2*N], N, A_nrows-A2, B_nrows-B2, depth+1);
32     }
33  }
34
35  cilk void trans_diagonal (val_t *A, long N/*rowlen*/, long size, int depth) {
36     if (size<=BASE) {
37         long i,j;
38         for (i=1; i<size; i++) {
39             for (j=0; j<i; j++) {
40                 val_t Aij = A[i+j*N];
41                 val_t Aji = A[j+i*N];
42                 A[i+j*N] = Aji;
43                 A[j+i*N] = Aij;
44             }
45         }
46     } else {
47         long s2 = size-size/2;
48         spawn trans_diagonal(A,           N, s2,      depth+1);
49         spawn trans_diagonal(&A[s2+N*s2], N, size-s2, depth+1);
50         spawn trans_offdiag (&A[0+N*s2], &A[s2+N*0], N, s2, size-s2, depth+1);
51     }
52  }
53
54  cilk void trans (val_t *A, long N) {
55     spawn trans_diagonal(A, N, N, 0);
56  }
```

```
57
58  cilk void initialize_fft_data(fftw_complex * arr, long n)
59  {
60   if (n<16) {
61      long i;
62      for (i = 0; i < n; i++) { /* initialize to some arbitrary values: */
63          c_re(arr[i]) = 1-2*drand48();
64          c_im(arr[i]) = 1-2*drand48();
65      }
66   } else {
67     spawn initialize_fft_data(arr, n/2);
68     spawn initialize_fft_data(arr+n/2, n-n/2);
69   }
70  }
71
72  inline void cmul3 (fftw_complex *dest, fftw_complex *a, fftw_complex *b, fftw_complex *c) {
73     double a_re = c_re(*a);
74     double a_im = c_im(*a);
75     double b_re = c_re(*b);
76     double b_im = c_im(*b);
77     double ab_re = a_re*b_re - a_im*b_im;
78     double ab_im = a_re*b_im + b_re*a_im;
79     double c_re = c_re(*c);
80     double c_im = c_im(*c);
81     c_re(*dest) = ab_re*c_re - ab_im*c_im;
82     c_im(*dest) = ab_re*c_im + c_re*ab_im;
83  }
84
85  /* How to improve the computation twiddle factors:
86   *  Done naivly, the twiddle factors, which use sin() and cos()
87   *  are expensive to compute.
88   *
89   *  Storing an O(N) table is too costly.  The standard solution is store
90   *  two O(sqrt(N)) tables of sin/cos values.  Specifically, to compute
91   *  w^j, write j=j0*sqrtn+j1, 0<=j0,j1<sqrtn, and compute w^(j0*sqrtn) *
92   *  w^j1.  The two operands come from a table lookup.  The loss of
93   *  accuracy is negligible.
94   */
95
96  // These tables are of length sqrtn
97  fftw_complex *w_to_i;               /* w_to_i[j] == e^{-2*M_PI*sqrt(-1)*j/(sqrtn*sqrtn)}     0<=i<=sqrtn */
98  fftw_complex *w_to_sqrtn_to_i;   /* w_to_i[j] == e^{-2*M_PI*sqrt(-1)*sqrtn/(sqrtn*sqrtn)}    0<=i<=sqrtn */
99
100 cilk void twiddle_iter (fftw_complex *a, int sqrtn, int low, int hi) {
101    if (low+1==hi) {
102        int i=low;
103        int j;
104        long isqrtn = (long)i*(long)sqrtn;
105        int i0 = i/sqrtn;
106        int i1 = i%sqrtn;
107        int ij0 = 0;
108        int ij1 = 0;
109        assert(0 <= ij0 && ij0 < sqrtn);
110        assert(0 <= ij1 && ij1 < sqrtn);
111        for (j=0; j<sqrtn; j++) {
112            long i_j = isqrtn + j;
113            cmul3(&a[i_j], &a[i_j], &w_to_i[ij1], &w_to_sqrtn_to_i[ij0]);
114            assert(i*j==ij0*sqrtn+ij1);
```

```
115             ij0 += i0;
116             ij1 += i1;
117             if (ij1>=sqrtn) { ij1-=sqrtn; ij0++; assert(ij1<sqrtn); }
118             ij0 %= sqrtn;
119         }
120     } else {
121         spawn twiddle_iter(a, sqrtn, low, (low+hi)/2);
122         spawn twiddle_iter(a, sqrtn, (low+hi)/2, hi);
123     }
124 }
125 
126 cilk void multiply_by_twiddle_factors (fftw_complex *a, int sqrtn)
127         /* Multiply twiddle factors. */
128         /*   a[i,j] *= exp(-2*M_PI*sqrt(-1)*sqrtn*j/(sqrtn*sqrtn)) */
129 {
130     spawn twiddle_iter(a, sqrtn, 0, sqrtn);
131 }
132 
133 cilk void do_fft (fftw_plan sqrtn_plan, fftw_complex * cin, int sqrtn) {
134     spawn fftw_cilk(sqrtn_plan, sqrtn, cin, 1, sqrtn, NULL, 1, sqrtn);
135     sync;
136 
137     spawn multiply_by_twiddle_factors(cin, sqrtn);
138     sync;
139 
140     spawn trans(cin, sqrtn);
141     sync;
142 
143     spawn fftw_cilk(sqrtn_plan, sqrtn, cin, 1, sqrtn, NULL, 1, sqrtn);
144     sync;
145 }
146 
147 cilk int main(int argc, char **argv)
148 {
149     Cilk_time start_t, end_t;
150     fftw_complex *cin;
151     double time1;
152     fftw_plan plan;
153     long n, sqrtn;
154 
155     sqrtn = strtol(argv[1], 0, 10);
156     printf("P=%d n=%ld\n", Cilk_active_size, n);
157     n = sqrtn*sqrtn;
158     assert(n>0);
159 
160     w_to_i          = malloc(sizeof(*w_to_i)*sqrtn);
161     w_to_sqrtn_to_i = malloc(sizeof(*w_to_i)*sqrtn);
162     { /* Initialize twiddles */
163         int i;
164         for (i=0; i<sqrtn; i++) {
165             {
166                 double phi = -2*M_PI*(double)i/((double)sqrtn*(double)sqrtn);
167                 c_re(w_to_i[i]) = cos(phi);
168                 c_im(w_to_i[i]) = sin(phi);
169             }
170             {
171                 double phi = -2*M_PI*(double)i*(double)sqrtn/((double)sqrtn*(double)sqrtn);
172                 c_re(w_to_sqrtn_to_i[i]) = cos(phi);
```

```
                c_im(w_to_sqrtn_to_i[i]) = sin(phi);
            }
        }
    }

    plan = fftw_create_plan(sqrtn, FFTW_FORWARD, FFTW_MEASURE|FFTW_USE_WISDOM|FFTW_IN_PLACE);

    cin = malloc(n * sizeof(fftw_complex));
    assert(cin);
    spawn initialize_fft_data(cin, sqrtn);  /* Do in parallel: Try to do "first touch" in many places */
    sync;
    start_t = Cilk_get_wall_time();
    spawn do_fft(plan, cin, sqrtn);
    sync;
    end_t = Cilk_get_wall_time();
    time1=Cilk_wall_time_to_sec(end_t - start_t);
    printf("P=%d n=%ld ", Cilk_active_size, n);
    printf(" time=%gs  %g Gflop/s\n", time1, 5e-9*n*(log((double)n)/log(2.0))/time1);

    fftw_destroy_plan(plan);
    free(cin);

    return 0;
}
```

F.2 transpose-base.c

```
/* Copyright (c) 2006 Bradley C. Kuszmaul */
#include <fftw.h>
#include "transpose-base.h"

void tran_base (val_t *A, val_t *B, long N, long A_nrows, long B_nrows) {
    long i,j;
    val_t tmp[BASE][BASE];
    /* The problem with the simpler code is that performance suffers when
     * there are associativity conflicts, which is the common case when the stride is 2^k.
     * I know of no cache-oblivious way to avoid this problem.  The best known
     * solution is to stop the recursion at Theta(sizeof(L1))-size matrices.  (On Itanium, it may be
     * size of (L2)), copy the input leaf matrix into an intermediate buffer of small stride, and copy
     * the buffer into the output leaf matrix. */
    for (j=0; j<B_nrows; j++) {
        for (i=0; i<A_nrows; i++) {
            tmp[i][j] = A[i+j*N];
        }
    }
    for (i=0; i<A_nrows; i++) {
        for (j=0; j<B_nrows; j++) {
            val_t Bji = B[j+i*N];
            B[j+i*N] = tmp[i][j];
            tmp[i][j] = Bji;
        }
    }
    for (j=0; j<B_nrows; j++) {
        for (i=0; i<A_nrows; i++) {
            A[i+j*N] = tmp[i][j];
        }
    }
}
```

F.3 transpose-base.h

```
1   typedef fftw_complex val_t;
2   #define BASE 8
3   void tran_base (val_t *A, val_t *B, long N, long A_nrows, long B_nrows);
```