LISTING PROGRAM PERKALIAN MATRIKS
DENGAN MPICH2

```
/* -------------------------------------------------------------
--------- */
/* -- Parallel Matrix Operations - Matrix Multiplication
-- */
/* --
-- */
/* -- E. van den Berg
02/10/2001 -- */
/* -------------------------------------------------------------
--------- */

#include "mpi.h"
#include <stdio.h>
#include <stdlib.h>

#define TAG_MATRIX_PARTITION  0x4560

// Type definitions
typedef struct
{  unsigned int   m, n;  // Rows, cols
   double        *data;  // Data, ordered by row, then by col
   double       **rows;  // Pointers to rows in data
} TMatrix;

// Global variables
int processor_rank  = 0;
int processor_count = 1;

// Function declaractions
TMatrix createMatrix     (const unsigned int rows, const unsigned
int cols);
void    freeMatrix       (TMatrix *matrix);
int     validMatrix      (TMatrix matrix);
TMatrix initMatrix       (void);
TMatrix matrixMultiply   (TMatrix A, TMatrix B);
void    doMatrixMultiply (TMatrix A, TMatrix B, TMatrix C);
void    printMatrix      (TMatrix A);


/* -------------------------------------------------------------
--------- */
int main (int argc, char *argv[])
/* -------------------------------------------------------------
--------- */
{  MPI_Status   status;
   TMatrix      A,B,C,D;
   unsigned int m, n, i, j, offset;
   double       time0, time1;

   A = initMatrix(); B = initMatrix(); C = initMatrix(); D =
initMatrix();

   // Always use MPI_Init first
```

```
   if (MPI_Init(&argc, &argv) == MPI_SUCCESS) do
   {  // Determine which number is assigned to this processor
      MPI_Comm_size (MPI_COMM_WORLD, &processor_count);
      MPI_Comm_rank (MPI_COMM_WORLD, &processor_rank );

      if (processor_rank == 0)
      {  printf ("Please enter matrix dimension n : ");
scanf("%u", &n);

        // Record starting time
        time0 = MPI_Wtime();

        // Allocate memory for matrices
        A = createMatrix(n, n);
       B = createMatrix(n, n);
       C = createMatrix(n, n);

       // Initialize matrices
       for (i = 0; i < n ; i++)
       {  for (j = 0; j < n; j++)
          {  A.rows[i][j] = (int)(3.0 * (rand() / (RAND_MAX +
1.0)));
             B.rows[i][j] = (int)(3.0 * (rand() / (RAND_MAX +
1.0)));
          }
       }
       if (!validMatrix(A) || !validMatrix(B) || !validMatrix(C))
n = 0;

        // Broadcast (send) size of matrix
        MPI_Bcast((void *)&n, 1, MPI_INT, 0, MPI_COMM_WORLD); if
(n == 0) break;
        m = n / processor_count;

        // Broadcast (send) B matrix
        MPI_Bcast((void *)B.data, n*n, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

        // Send each process it's own part of A
       offset = n - m * (processor_count - 1);
       for (i = 1; i < processor_count; i++)
       {  MPI_Send((void *)A.rows[offset], m*n, MPI_DOUBLE,
                   i, TAG_MATRIX_PARTITION, MPI_COMM_WORLD);
          offset += m;
       }

       // Multiply own part of matrix A with B into already
existing matrix C
       A.m = n - m * (processor_count - 1);
       doMatrixMultiply(A,B,C);
       A.m = n;

       // Receive part of C matrix from each process
       offset = n - m * (processor_count - 1);
       for (i = 1; i < processor_count; i++)
       {  MPI_Recv((void *)C.rows[offset], m*n, MPI_DOUBLE,
                   i, TAG_MATRIX_PARTITION, MPI_COMM_WORLD,
&status);
          offset += m;
```

```
       }

       // Record finish time
       time1 = MPI_Wtime();

         // Print time statistics
       printf ("Total time using [%2d] processors : [%f]
seconds\n", processor_count, time1 - time0);

      }
      else
      {
         // Broadcast (receive) size of matrix
         MPI_Bcast((void *)&n, 1, MPI_INT, 0, MPI_COMM_WORLD); if
(n == 0) break;

         // Allocate memory for matrices
         m = n / processor_count;
       A = createMatrix(m, n);
       B = createMatrix(n ,n);

         // Broadcast (receive) B matrix
         MPI_Bcast((void *)B.data, n*n, MPI_DOUBLE, 0,
MPI_COMM_WORLD);
       MPI_Recv((void *)A.data, m*n, MPI_DOUBLE, 0,
TAG_MATRIX_PARTITION, MPI_COMM_WORLD, &status);

       // Multiply local matrices
       C = matrixMultiply(A,B);

       // Send back result
       MPI_Send((void *)C.data, m*n, MPI_DOUBLE, 0,
TAG_MATRIX_PARTITION, MPI_COMM_WORLD);
      }
   } while (0); // Enable break statement to be used above

   // Free matrix data
   freeMatrix(&A); freeMatrix(&B); freeMatrix(&C);

   // Wait for everyone to stop
   MPI_Barrier(MPI_COMM_WORLD);

   // Always use MPI_Finalize as the last instruction of the
program
   MPI_Finalize();

   return 0;
}

/* ----------------------------------------------------------------
--------- */
TMatrix createMatrix(const unsigned int rows, const unsigned int
cols)
/* ----------------------------------------------------------------
--------- */
{  TMatrix           matrix;
   unsigned long int m, n;
   unsigned int      i;
```

```
   m = rows; n = cols;
   matrix.m    = rows;
   matrix.n    = cols;
   matrix.data = (double *) malloc(sizeof(double) * m * n);
   matrix.rows = (double **) malloc(sizeof(double *) * m);
   if (validMatrix(matrix))
   {  matrix.m = rows;
      matrix.n = cols;
      for (i = 0; i < rows; i++)
      {  matrix.rows[i] = matrix.data + (i * cols);
      }
   }
   else
   {  freeMatrix(&matrix);
   }

   return matrix;
}


/* ------------------------------------------------------------------
--------- */
void freeMatrix (TMatrix *matrix)
/* ------------------------------------------------------------------
--------- */
{  if (matrix == NULL) return;

   if (matrix -> data) { free(matrix -> data); matrix -> data =
NULL; }
   if (matrix -> rows) { free(matrix -> rows); matrix -> rows =
NULL; }
   matrix -> m = 0;
   matrix -> n = 0;
}


/* ------------------------------------------------------------------
--------- */
int validMatrix (TMatrix matrix)
/* ------------------------------------------------------------------
--------- */
{  if ((matrix.data == NULL) || (matrix.rows == NULL) ||
       (matrix.m == 0) || (matrix.n == 0))
        return 0;
   else return 1;
}

/* ------------------------------------------------------------------
--------- */
TMatrix initMatrix()
/* ------------------------------------------------------------------
--------- */
{  TMatrix matrix;

   matrix.m = 0;
   matrix.n = 0;
   matrix.data = NULL;
   matrix.rows = NULL;
```

```
   return matrix;
}


/* ---------------------------------------------------------------
--------- */
TMatrix matrixMultiply(TMatrix A, TMatrix B)
/* ---------------------------------------------------------------
--------- */
{  TMatrix C;

   C = initMatrix();

   if (validMatrix(A) && validMatrix(B) && (A.n == B.m))
   {  C = createMatrix(A.m, B.n);
      if (validMatrix(C))
      {  doMatrixMultiply(A, B, C);
      }
   }

   return C;
}


/* ---------------------------------------------------------------
--------- */
void doMatrixMultiply(TMatrix A, TMatrix B, TMatrix C)
/* ---------------------------------------------------------------
--------- */
{  unsigned int i, j, k;
   double sum;

   for (i = 0; i < A.m; i++) // Rows
   {  for (j = 0; j < B.n; j++) // Cols
      {  sum = 0;
         for (k = 0; k < A.n; k++) sum += A.rows[i][k] *
B.rows[k][j];
         C.rows[i][j] = sum;
      }
   }
}


/* ---------------------------------------------------------------
--------- */
void printMatrix(TMatrix A)
/* ---------------------------------------------------------------
--------- */
{  unsigned int i, j;

   if (validMatrix(A))
   {  for (i = 0; i < A.m; i++)
      {  for (j = 0; j < A.n; j++) printf ("%7.3f ",
A.rows[i][j]);
       printf ("\n");
      }
   }
}
```

Lampiran II

LISTING PROGRAM PERKALIAN MATRIKS DENGAN CILK++

```
/*
 * matrix-mulitpy.cilk
 *
 * First of five matrix multiply examples to compare dense matrix
multiplication
 * algorithms using Cilk parallelization.
 *   Example 1: Straightforward loop parallelization of matrix
multiplication.
 *
 * Copyright (c) 2007-2008 Cilk Arts, Inc.  55 Cambridge Street,
 * Burlington, MA 01803.  Patents pending.  All rights reserved.
You may
 * freely use the sample code to guide development of your own
works,
 * provided that you reproduce this notice in any works you make
that
 * use the sample code.  This sample code is provided "AS IS"
without
 * warranty of any kind, either express or implied, including but
not
 * limited to any implied warranty of non-infringement,
merchantability
 * or fitness for a particular purpose.  In no event shall Cilk
Arts,
 * Inc. be liable for any direct, indirect, special, or
consequential
 * damages, or any other damages whatsoever, for any use of or
reliance
 * on this sample code, including, without limitation, any lost
 * opportunity, lost profits, business interruption, loss of
programs or
 * data, even if expressly advised of or otherwise aware of the
 * possibility of such damages, whether in an action of contract,
 * negligence, tort, or otherwise.
 *
 */

#include <cilk.h>
#include <cilkview.h>
#include <iostream>

#define DEFAULT_MATRIX_SIZE 4

// Multiply double precision square n x n matrices. A = B * C
// Matrices are stored in row major order.
// A is assumed to be initialized.
void matrix_multiply(double* A, double* B, double* C, unsigned int
n)
{
    if (n < 1) {
        return;
    }
```

```
    cilk_for(unsigned int i = 0; i < n; ++i) {
        // This is the only Cilk++ keyword used in this program
            // Note the order of the loops and the code motion of
the i * n and k * n
            // computation. This gives a 5-10 performance
improvement over exchanging
            // the j and k loops.
            int itn = i * n;
        for (unsigned int k = 0; k < n; ++k) {
            for (unsigned int j = 0; j < n; ++j) {
              int ktn = k * n;
                // Compute A[i,j] in the innner loop.
                A[itn + j] += B[itn + k] * C[ktn + j];
            }
        }
    }
    return;
}

void print_matrix(double* M, int nn)
{
    for (int i = 0; i < nn; ++i) {
        for (int j = 0; j < nn; ++j) {
            std::cout << M[i * nn + j] << ",  ";
        }
        std::cout << std::endl;
    }
    return;
}

int cilk_main(int argc, char** argv) {
    // Create random input matrices. Override the default size
with argv[1]
    // Warning: Matrix indexing is 0 based.
    int nn = DEFAULT_MATRIX_SIZE;
    if (argc > 1) {
        nn = std::atoi(argv[1]);
    }

    std::cout << "Simple algorithm: Mulitply two " << nn << " by "
<< nn
        << " matrices, coumputing A = B*C" << std::endl;

    double* A = (double*) calloc(nn* nn, sizeof(double));
    double* B = (double*) calloc(nn* nn, sizeof(double));
    double* C = (double*) calloc(nn* nn, sizeof(double));
    if (NULL == A || NULL == B || NULL == C) {
        std::cout << "Fatal Error. Cannot allocate matrices A, B,
and C."
            << std::endl;
        return 1;
    }

    // Populate B and C pseudo-randomly -
    // The matrices are populated with random numbers in the range
(-1.0, +1.0)
    cilk_for(int i = 0; i < nn * nn; ++i) {
        B[i] = (float) ((i * i) % 1024 - 512) / 512;
    }
```

```
    cilk_for(int i = 0; i < nn * nn; ++i) {
        C[i] = (float) (((i + 1) * i) % 1024 - 512) / 512;
    }

    // Multiply to get A = B*C
    cilk::cilkview cv;
    cv.reset();
    cv.start();
    matrix_multiply(A, B, C, (unsigned int)nn);
    cv.stop();
    float par_time = cv.accumulated_milliseconds() / 1000.f;
    std::cout << " Matrix Multiply took " << par_time << "
seconds."
        << std::endl;

    // If n is small, print the results
    if (nn <= 8) {
        std::cout << "Matrix A:" << std::endl;
        print_matrix(B, nn);
        std::cout << std::endl << "Matrix B:" << std::endl;
        print_matrix(C, nn);
        std::cout << std::endl << "Matrix C = A * B:" <<
std::endl;
        print_matrix(A, nn);
    }

    free(A);
    free(B);
    free(C);
    return 0;
}
```

Lampiran III

LISTING PROGRAM *SORTING* DENGAN M PICH2

```
/* ------------------------------------------------------------
--------- */
/* -- Parallel Binary Sorting Example
-- */
/* --
-- */
/* -- E. van den Berg
01/10/2001 -- */
/* ------------------------------------------------------------
--------- */

#include <stdio.h>
#include <stdlib.h>
#include "mpi.h"

#define TAG_HEADER          0x1234
#define TAG_DATA       0x1235
#define TAG_SORTED_DATA     0x1236

// Type definitions
typedef struct
{  int       processor_rank;
   int       processor_count;
   int       processor_owner;
   int      *data;
   long int  offset;
   long int  size;
} TParallelSortHeader;


// Global variables
int processor_rank  = 0;
int processor_count = 1;


// Function declarations
void parallelBinarySort   (int *data, long int size);
void doParallelBinarySort (TParallelSortHeader headerA, int
hasOwner);
void divideWork           (TParallelSortHeader *headerA,
TParallelSortHeader *headerB);
void merge                (int *sortedA, long int sizeA, int
*sortedB, long int sizeB);
void sequentialBinarySort (int *data, long int size);



/* ------------------------------------------------------------
--------- */
int main (int argc, char *argv[])
/* ------------------------------------------------------------
--------- */
{  double    time0, time1;
   long int  size, i;
```

```
   int      *data = NULL;

   // Always use MPI_Init first
   if (MPI_Init(&argc, &argv) == MPI_SUCCESS)
   {  // Determine which number is assigned to this processor
      MPI_Comm_size (MPI_COMM_WORLD, &processor_count);
      MPI_Comm_rank (MPI_COMM_WORLD, &processor_rank );

      if (processor_rank == 0)
      {  // Read data
         printf ("Number of elements : "); scanf("%lu", &size);

         // Fill data array with random numbers
         data = (int *) malloc (sizeof(int) * size);
         for (i = 0; i < size; i++) data[i] = (int)(3000.0 *
(rand() / (RAND_MAX + 1.0)));
      }

      time0 = MPI_Wtime();
      {
         parallelBinarySort(data, size);
      }
      time1 = MPI_Wtime();


      if (processor_rank == 0)
      {
       printf ("Total time : %f seconds\n", time1 - time0);
      }
   }

   // Wait for every process to finish
   MPI_Barrier(MPI_COMM_WORLD);

   // Always use MPI_Finalize as the last MPI instruction of the
program!
   MPI_Finalize();

   return 0;
}



/* ----------------------------------------------------------------
--------- */
void parallelBinarySort (int *data, long int size)
/* ----------------------------------------------------------------
--------- */
{  TParallelSortHeader headerA;
   MPI_Status          status;

   if (processor_rank == 0)
   {  if (data == NULL) size  = 1;
      headerA.processor_rank  = 0;
      headerA.processor_count = processor_count;
      headerA.processor_owner = -1;
      headerA.data            = NULL;
      headerA.size            = 0;
      headerA.offset          = 0;
```

```
      // Send bogus header message to all processor that will not
be used
      // due to a shortage of data items. Otherwise the processors
      // would wait for a header message forever.
      while (headerA.processor_count > size)
      {  headerA.processor_count --;
         MPI_Send((void *)&headerA, sizeof(headerA), MPI_BYTE,
                  headerA.processor_count, TAG_HEADER,
MPI_COMM_WORLD);
      }

      // Set data array field
      if (data == NULL) return;
      headerA.data = data;
      headerA.size = size;

      // Initiate real parallel sort routine
      doParallelBinarySort (headerA, 0);
   }
   else
   {
      // Receive header
      MPI_Recv((void *)&headerA, sizeof(headerA), MPI_BYTE,
               MPI_ANY_SOURCE, TAG_HEADER, MPI_COMM_WORLD,
&status);

      // If no elements will be sent to this process, cancel for
this process.
      if (headerA.size == 0) { printf ("[%d] Cancelled.\n",
processor_rank); return; }

      // This malloc might fail and solving it in this case
      // would require sending a message to the 'parent' to
      // indicate that the operation be cancelled. If we just
      // exit here, then the other processor will be hanging,
      // waiting for the message to be received.
      headerA.data = (int *) malloc (sizeof(int) * headerA.size);

      // Receive the data to be sorted
      MPI_Recv((void *)headerA.data, headerA.size, MPI_INT,
               headerA.processor_owner, TAG_DATA, MPI_COMM_WORLD,
&status);

      // Initiate real parallel sort routine
      doParallelBinarySort (headerA, 1);
   }
}


/* ----------------------------------------------------------------
--------- */
void doParallelBinarySort (TParallelSortHeader headerA, int
hasOwner)
/* ----------------------------------------------------------------
--------- */
{  TParallelSortHeader headerB;
   MPI_Status          status;
   int                *dataB;
```

```
   // Divide work specified by header
   divideWork (&headerA, &headerB);

   // Send data to other process
   if (headerB.size > 0)
   {  // Set data field in headerB to NULL and send header
      // to prevent it from being used directly by the other
      // process. Not really necessary but prevents difficult
      // to find bugs.
      dataB = headerB.data; headerB.data = NULL;
      MPI_Send((void *)&headerB, sizeof(headerB), MPI_BYTE,
               headerB.processor_rank,
               TAG_HEADER, MPI_COMM_WORLD);

      // Restore data field in headerB and send data
      headerB.data = dataB;
      MPI_Send((void *)headerB.data, headerB.size, MPI_INT,
               headerB.processor_rank,
               TAG_DATA, MPI_COMM_WORLD);
   }

   // Split data if there is (A) sufficient data, and (B)
processors available.
   // Otherwise just initiate sequential binary sort within this
process.
   if ((headerA.size > 1) && (headerA.processor_count > 1))
   {  doParallelBinarySort(headerA, 0);
   }
   else
   {  sequentialBinarySort(headerA.data, headerA.size);
   }

   // Data provided by headerA has now been sorted. If any
expected,
   // receive data as specified by headerB.
   if (headerB.size > 0)
   {  MPI_Recv((void *)headerB.data, headerB.size, MPI_INT,
               headerB.processor_rank,
               TAG_SORTED_DATA, MPI_COMM_WORLD, &status);
   }

   // Merge results
   merge(headerA.data, headerA.size, headerB.data, headerB.size);

   // Send back result if has owner and free memory
   if (hasOwner)
   {  MPI_Send((void *)headerA.data, headerA.size + headerB.size,
MPI_INT,
               headerA.processor_owner,
               TAG_SORTED_DATA, MPI_COMM_WORLD);

      // Free allocated memory
      free(headerA.data);
   }
}
```

```
/* ----------------------------------------------------------------
--------- */
void divideWork (TParallelSortHeader *headerA, TParallelSortHeader
*headerB)
/* ----------------------------------------------------------------
--------- */
{  int countA, countB, sizeA, sizeB;

   // Is there a processor available to divide work to
   if (headerA -> processor_count > 1)
   {
      countA = headerA -> processor_count / 2;
      countB = headerA -> processor_count - countA;
      sizeA  = (int)(((headerA -> size) / (double)(countA +
countB)) * countA);
      sizeB  = headerA -> size - sizeA;
      headerA -> processor_count = countA;
      headerA -> size            = sizeA;
      headerB -> processor_rank  = headerA -> processor_rank +
countA;
      headerB -> processor_count = countB;
      headerB -> processor_owner = headerA -> processor_rank;
      headerB -> data            = headerA -> data + sizeA;
      headerB -> size            = sizeB;
      headerB -> offset          = headerA -> offset + sizeA;
   }
   else
   {
      headerB -> processor_rank  = -1;
      headerB -> processor_count = -1;
      headerB -> processor_owner = -1;
      headerB -> data            = NULL;
      headerB -> size            = 0;
      headerB -> offset          = -1;
   }
}


/* ----------------------------------------------------------------
--------- */
void merge (int *sortedA, long int sizeA, int *sortedB, long int
sizeB)
/* ----------------------------------------------------------------
--------- */
{  int *sortedC, sizeC;
   long int i = 0, j = 0, k = 0;

   // Allocate memory for temporary array
   sizeC   = sizeA + sizeB;
   sortedC = (int *) malloc (sizeof(int) * sizeC);

   // If memory allocation for temporary result array successful,
   // start merging. After merging into C, the results will be
   // written back to A and B.
   if (sortedC)
   {
      // While not at the end of one of the sequences, determine
the
```

```
      // smaller element of the two currently pointed-to elements
in
      // the sequences and copy this element into sequence C.
      while ((i < sizeA) && (j < sizeB))
      {  if ((sortedA)[i] <= (sortedB)[j])
            sortedC[k++] = (sortedA)[i++];
         else sortedC[k++] = (sortedB)[j++];
      }

      // While not at the end of A, copy remaining elements to C.
      while (i < sizeA) sortedC[k++] = (sortedA)[i++];

      // While not at the end of B, copy remaining elements to C.
      while (j < sizeB) sortedC[k++] = (sortedB)[j++];

      // Copy back the first half of C into A, and second half to
B.
      // The halves are determined by the number of elements in A
and B.
      k = 0;
      for (i = 0; i < sizeA; i++) sortedA[i] = sortedC[k++];
      for (j = 0; j < sizeB; j++) sortedB[j] = sortedC[k++];

      // Free memory for temporary array
      free(sortedC);
   }
}


/* ----------------------------------------------------------------
--------- */
void sequentialBinarySort (int *data, long int size)
/* ----------------------------------------------------------------
--------- */
{  int *dataA, *dataB;
   long int sizeA, sizeB;

   if (size > 1)
   {  dataA = data;
      sizeA = size / 2;
      dataB = data + sizeA;
      sizeB = size - sizeA;
      sequentialBinarySort(dataA, sizeA);
      sequentialBinarySort(dataB, sizeB);
      merge(dataA, sizeA, dataB, sizeB);
   }
}
```

Lampiran IV

LISTING PROGRAM *SORTING* DENGAN CILK++

```
// -*- C++ -*-
/*
 * qsort.cilk
 *
 * An implementation of quicksort using Cilk parallelization.
 *
 * Copyright (c) 2007-2008 Cilk Arts, Inc.  55 Cambridge Street,
 * Burlington, MA 01803.  Patents pending.  All rights reserved.
You may
 * freely use the sample code to guide development of your own
works,
 * provided that you reproduce this notice in any works you make
that
 * use the sample code.  This sample code is provided "AS IS"
without
 * warranty of any kind, either express or implied, including but
not
 * limited to any implied warranty of non-infringement,
merchantability
 * or fitness for a particular purpose.  In no event shall Cilk
Arts,
 * Inc. be liable for any direct, indirect, special, or
consequential
 * damages, or any other damages whatsoever, for any use of or
reliance
 * on this sample code, including, without limitation, any lost
 * opportunity, lost profits, business interruption, loss of
programs or
 * data, even if expressly advised of or otherwise aware of the
 * possibility of such damages, whether in an action of contract,
 * negligence, tort, or otherwise.
 *
 */
#include <cilk.h>
#include <cilkview.h>
#include <algorithm>
#include <iostream>
#include <iterator>
#include <functional>

// Sort the range between bidirectional iterators begin and end.
// end is one past the final element in the range.
// Use the Quick Sort algorithm, using recursive divide and
conquer.
// This function is NOT the same as the Standard C Library qsort()
function.
// This implementation is pure C++ code before Cilk++ conversion.

void sample_qsort(int * begin, int * end)
{
    if (begin != end) {
        --end;  // Exclude last element (pivot) from partition
        int * middle = std::partition(begin, end,
                          std::bind2nd(std::less<int>(), *end));
```

```
        using std::swap;
        swap(*end, *middle);    // move pivot to middle
        cilk_spawn sample_qsort(begin, middle);
        sample_qsort(++middle, ++end); // Exclude pivot and
restore end
        cilk_sync;
    }
}

// A simple test harness
int qmain(int n)
{
    int* a = new int[n];
    cilk::cilkview cv;

    for (int i = 0; i < n; ++i)
        a[i] = i;
    std::random_shuffle(a, a + n);
    std::cout << "Sorting " << n << " integers" << std::endl;
    cv.start();
    sample_qsort(a, a + n);
    cv.stop();
    cv.dump("qsort-results", false);

    std::cout << cv.accumulated_milliseconds() / 1000.f << "
seconds" << std::endl;

    // Confirm that a is sorted and that each element contains the
index.
    for (int i = 0; i < n - 1; ++i) {
        if (a[i] >= a[i + 1] || a[i] != i) {
            std::cout << "Sort failed at location i=" << i << "
a[i] = "
                      << a[i] << " a[i+1] = " << a[i + 1] <<
std::endl;
            delete[] a;
            return 1;
        }
    }
    std::cout << "Sort succeeded." << std::endl;
    delete[] a;
    return 0;
}

int cilk_main(int argc, char* argv[])
{
    int n = 10 * 1000 * 1000;
    if (argc > 1) {
         n = std::atoi(argv[1]);
         if (n <= 0) {
              std::cerr << "Invalid argument" << std::endl;
              std::cerr << "Usage: qsort N" << std::endl;
              std::cerr << "       N = number of elements to sort"
<< std::endl;
              return 1;
         }
    }
    return qmain(n);
}
```