Parallel Patternlets

CSinParallel Project

January 21, 2013

CONTENTS

This document contains simple examples of basic elements that are combined to form patterns often used in programs employing parallelism. The examples are separated between two major *coordination patterns*:

1. message passing used on single multiprocessor machines or clusters of distributed computers, and
2. mutual exclusion between threads executing concurrently on a single shared memory system.

Both sets of examples are illustrated with the C programming language, using standard popular available libraries. The message passing example uses a C library called MPI (Message Passing Interface). The mutual Exclusion/shared memory examples use the OpenMP library.

CHAPTER

ONE

SOURCE CODE

Please download all examples from this tarball: `patternlets.tgz`

CHAPTER

TWO

PATTERNLET EXAMPLES

2.1 Message Passing Parallel Patternlets

Parallel programs contain *patterns*: code that recurs over and over again in solutions to many problems. The following examples show very simple examples of small portions of these patterns that can be combined to solve a problem. These C code examples use the Message Passing Interface (MPI) library, which is suitable for use on either a single pultiprocessor machine or a cluster of machines.

2.1.1 Source Code

Please download all examples from this tarball: `patternlets.tgz`

A C code file for each example below can be found in subdirectories of the MPI directory, along with a makefile and an example of how to execute the program.

2.1.2 0. Hello, World

First let us illustrate the basic components of an MPI program.

```
/* hello.c
 * ... illustrates the use of the basic MPI commands...
 * Joel Adams, Calvin College, at November 2009.
 */

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

int main(int argc, char** argv) {
        int id = -1, numProcesses = -1, length = -1;
        char myHostName[MPI_MAX_PROCESSOR_NAME];

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
        MPI_Get_processor_name (myHostName, &length);


        printf("Greetings from process %d of %d on %s\n",
                id, numProcesses, myHostName);

        MPI_Finalize();
```

```
        return 0;
}
```

2.1.3 1. The Master-Worker Implementation Strategy Pattern

```
/* masterServer.c
 * ... illustrates the basic master-worker pattern in MPI ...
 * Joel Adams, Calvin College, at November 2009.
 */

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

int main(int argc, char** argv) {
  int id = -1, numWorkers = -1, length = -1;
  char hostName[MPI_MAX_PROCESSOR_NAME];

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &numWorkers);
  MPI_Get_processor_name (hostName, &length);

  if ( id == 0 ) {  // process 0 is the master
    printf("Greetings from the master, # %d (%s) of %d processes\n",
           id, hostName, numWorkers);
  } else {          // processes with ids > 0 are workers
    printf("Greetings from a worker, # %d (%s) of %d processes\n",
           id, hostName, numWorkers);
  }

  MPI_Finalize();
  return 0;
}
```

2.1.4 2. Send-Receive (basic message passing pattern)

```
/* sendRecv.c
 * ... illustrates the use of the MPI_Send() and MPI_Recv() commands...
 * Joel Adams, Calvin College, at November 2009.
 */

#include <stdio.h>
#include <mpi.h>
#include <math.h>   // sqrt()

int odd(int number) { return number % 2; }

int main(int argc, char** argv) {
        int id = -1, numProcesses = -1;
        float sendValue = -1, receivedValue = -1;
        MPI_Status status;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
```

```
        if (numProcesses > 1 && !odd(numProcesses) ) {
                sendValue = sqrt(id);
                if ( odd(id) ) {  // odd processors send, then receive
                        MPI_Send(&sendValue, 1, MPI_FLOAT, id-1, 1, MPI_COMM_WORLD);
                        MPI_Recv(&receivedValue, 1, MPI_FLOAT, id-1, 2,
                                        MPI_COMM_WORLD, &status);
                } else {          // even processors receive, then send
                        MPI_Recv(&receivedValue, 1, MPI_FLOAT, id+1, 1,
                                        MPI_COMM_WORLD, &status);
                        MPI_Send(&sendValue, 1, MPI_FLOAT, id+1, 2, MPI_COMM_WORLD);
                }

                printf("Process %d of %d computed %f and received %f\n",
                        id, numProcesses, sendValue, receivedValue);
        } else if ( !id) {  // only process 0 does this part
                printf("\nPlease run this program using -np N where N is positive and even.\n\n");
        }

        MPI_Finalize();
        return 0;
}
```

2.1.5 3. Data Decomposition: on *slices* using parallel-for

In this example, the data being decomposed in simply the set of integeres from zero to 15, inclusive.

```
/* parallelForSlices.c
 * ... illustrates the parallel for loop pattern in MPI
 *        in which processes perform the loop's iterations in 'slices'
 *        (simple, and useful when loop iterations do not access
 *         memory/cache locations) ...
 * Joel Adams, Calvin College, at November 2009.
 */

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

int main(int argc, char** argv) {
        const int ITERS = 16;
        int id = -1, numProcesses = -1, i = -1;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

        for (i = id; i < ITERS; i += numProcesses) {
                printf("Process %d is performing iteration %d\n",
                        id, i);
        }

        MPI_Finalize();
        return 0;
}
```

2.1.6 4. Data Decomposition: on *blocks* using parallel-for

This is a basic example that does not yet include a data array, though it would typically be used when each process would be working on a portion of an array that could have been looped over in a sequential solution.

```
/* parallelForBlocks.c
 * ... illustrates the parallel for loop pattern in MPI
 *        in which processes perform the loop's iterations in 'blocks'
 *        (preferable when loop iterations do access memory/cache locations) ...
 * Joel Adams, Calvin College, at November 2009.
 */

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

int main(int argc, char** argv) {
        const int ITERS = 16;
        int id = -1, numProcesses = -1, i = -1,
                start = -1, stop = -1, blockSize = -1;

        MPI_Init(&argc, &argv);
        MPI_Comm_rank(MPI_COMM_WORLD, &id);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);

        blockSize = ITERS / numProcesses;      // integer division
        start = id * blockSize;                // find starting index
                                               // find stopping index
        if ( id < numProcesses - 1 ) {         // if not the last process
                stop = (id + 1) * blockSize;   //  stop where next process starts
        } else {                               // else
                stop = ITERS;                  //  last process does leftovers
        }

        for (i = start; i < stop; i++) {       // iterate through your range
                printf("Process %d is performing iteration %d\n",
                        id, i);
        }

        MPI_Finalize();
        return 0;
}
```

2.1.7 5. Broadcast: a special form of message passing

This example shows how to ensure that all processes have a copy of an array created by a single *master* node.

```
/* bcast.c
 * ... illustrates the use of MPI_Bcast()...
 * Joel Adams, Calvin College, at November 2009.
 */

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

// fill an array with values
void fill(int* a, int size) {
```

```
        int i;
        for (i = 0; i < size; i++) {
                a[i] = i+11;
        }
}

// display a string, a process id, and its array values
void print(char* str, int id, int* a) {
        printf("process %d %s: {%d, %d, %d, %d, %d, %d, %d, %d}\n",
           id, str, a[0], a[1], a[2], a[3], a[4], a[5], a[6], a[7]);
}

#define MAX 8

int main(int argc, char** argv) {
        int array[MAX] = {0};
        int numProcs, myRank;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

        if (myRank == 0) fill(array, MAX);

        print("array before", myRank, array);
        MPI_Bcast(array, MAX, MPI_INT, 0, MPI_COMM_WORLD);
        print("array after", myRank, array);

         MPI_Finalize();

        return 0;
}
```

2.1.8 6. Collective Communication: Reduction

Once processes have performed independent concurrent computations, possibly on some portion of decomposed data, it is quite commen to then *reduce* those individual computations into one value. This example shows a simple calculation done by each process being reduced to a sum and a maximum. In this example, MPI, has built-in computations, indicated by MPI_SUM and MPI_MAX in the following code.

```
/* reduce.c
 * ... illustrates the use of MPI_Reduce()...
 * Joel Adams, Calvin College, at November 2009.
 */

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

int main(int argc, char** argv) {
        int numProcs = -1, myRank = -1, square = -1, max = -1, sum = 0;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

        square = (myRank+1) * (myRank+1);
```

```
        printf("Process %d computed %d\n", myRank, square);

        MPI_Reduce(&square, &sum, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

        MPI_Reduce(&square, &max, 1, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD);

        if (myRank == 0) {
                printf("\nThe sum of the squares is %d\n\n", sum);
                printf("The max of the squares is %d\n\n", max);
        }

         MPI_Finalize();

        return 0;
}
```

2.1.9 7. Collective communication: Scatter for message-passing data decomposition

If processes can independently work on portions of a larger data array using the geometric data decomposition pattern, the scatter patternlet can be used to ensure that each process receives a copy of its portion of the array.

```
/* scatter.c
 * ... illustrates the use of MPI_Scatter()...
 * Joel Adams, Calvin College, at November 2009.
 */

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

int main(int argc, char** argv) {
        const int MAX = 8;
        int aSend[] = {11, 22, 33, 44, 55, 66, 77, 88};
        int* aRcv;
        int i, numProcs, myRank, numSent;

        MPI_Init(&argc, &argv);
        MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
        MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

        numSent = MAX / numProcs;
        aRcv = (int*) malloc( numSent * sizeof(int) );
        MPI_Scatter(aSend, numSent, MPI_INT, aRcv, numSent, MPI_INT, 0, MPI_COMM_WORLD);
        printf("Process %d: ", myRank);
        for (i = 0; i < numSent; i++) {
                printf(" %d", aRcv[i]);
        }
        printf("\n");

        free(aRcv);
         MPI_Finalize();

        return 0;
}
```

2.1.10 8. Collective communication: Gather for message-passing data decomposition

If processes can independently work on portions of a larger data array using the geometric data decomposition pattern, the gather patternlet can be used to ensure that each process sends a copy of its portion of the array back to the root, or master process.

```
/* gather.c
 * ... illustrates the use of MPI_Gather()...
 * Joel Adams, Calvin College, November 2009.
 */

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

int main(int argc, char** argv) {
   const int MAX = 3;
   int computeArray[MAX];
   int* gatherArray = NULL;
   int i, numProcs, myRank, totalGatheredVals;

   MPI_Init(&argc, &argv);
   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);

   for (i = 0; i < MAX; i++) {
      computeArray[i] = myRank * 10 + i;
   }

   totalGatheredVals = MAX * numProcs;
   gatherArray = (int*) malloc( totalGatheredVals * sizeof(int) );

   MPI_Gather(computeArray, MAX, MPI_INT,
              gatherArray, MAX, MPI_INT, 0, MPI_COMM_WORLD);

   if (myRank == 0) {
      for (i = 0; i < totalGatheredVals; i++) {
         printf(" %d", gatherArray[i]);
      }
      printf("\n");
   }

   free(gatherArray);
   MPI_Finalize();

   return 0;
}
```

2.2 Shared Memory Parallel Patternlets in OpenMP

When writing programs for shared-memory hardware with multiple cores, a programmer could use a low-level thread package, such as pthreads. An alternative is to use a compiler that processes OpenMP *pragmas*, which are compiler directives that enable the compiler to generate threaded code. Whereas pthreads uses an **explicit** multithreading mosel in which the programmer must explicitly create and manage threads, OpenMP uses an **implicit** multithreading model

in which the library handles thread creation and management, thus making the programmer's task much simpler and less error-prone.

The following are examples of C code with OpenMP pragmas The firrst three are basic illustrations to get used to the OpenMP pragmas. The rest illustrate how to implement particular patterns and what can go wrong when mutual exclusion is not properly ensured.

Note: by default OpenMP uses the **Thread Pool** pattern of concurrent execution. OpenMP programs initialze a group of threads to be used by a given program (often called a pool of threads). These threads will execute concurrently during portions of the code specified by the programmer.

2.2.1 Source Code

Please download all examples from this tarball: `patternlets.tgz`

A C code file for each example below can be found in subdirectories of the OpenMP directory, along with a makefile.

2.2.2 0. The OMP parallel pragma

The *omp parallel* pragma on line 18, when uncommented, tells the compiler to fork a set of threads to execute that particular line of code.

```
1  /* simpleParallel.c
2   * ... illustrates the use of OpenMP's parallel directive...
3   *
4   * Joel Adams, Calvin College, at November 2009.
5   *
6   * Usage: ./simpleParallel
7   * Compile & run, uncomment the pragma, recompile & run, compare.
8   */
9
10 #include <stdio.h>
11 #include <omp.h>
12 #include <stdlib.h>
13
14 int main(int argc, char** argv) {
15
16     printf("\n\nBefore...\n");
17
18 //    #pragma omp parallel
19     printf("\nDuring...");
20
21     printf("\n\nAfter...\n\n");
22
23     return 0;
24 }
```

2.2.3 1. Hello, World: default number of OpenMP threads

Note how there are OpenMP functions to obtain a thread number and the total number of threads. When the pragma is uncommented, note what the default number of threads is. Here the threads are forked and execute the block of code insode the curly braces on lines 18 through 20.

```c
/* parallelHello.c
 * ... illustrates the use of two basic OpenMP commands...
 *
 * Joel Adams, Calvin College, at November 2009.
 *
 * Usage: ./parallelHello
 * Compile & run, uncomment pragma, recompile & run, compare results
 */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {

//    #pragma omp parallel
    {
        int rank  = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        printf("Hello from thread %d of %d\n", rank, numThreads);
    }

    return 0;
}
```

2.2.4 2. Hello, World

Here we enter the number of threads to use on the command line.

```c
/* parallelHello2.c
 * ... illustrates the use of two basic OpenMP commands
 *         using the commandline to control the number of threads...
 *
 * Joel Adams, Calvin College, at November 2009.
 *
 * Usage: ./parallelHello2 [numThreads]
 * Compile & run with no commandline arg, rerun with different commandline args
 */



#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int rank, numThreads;

    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

    #pragma omp parallel
    {
        rank  = omp_get_thread_num();
        numThreads = omp_get_num_threads();
        printf("Hello from thread %d of %d\n", rank, numThreads);
```

```
    }

    return 0;
}
```

2.2.5 3. Master-Worker Implementation Strategy

```
/* masterWorker.c
 * ... illustrates the master-worker pattern in OpenMP
 *
 * Joel Adams, Calvin College, November 2009.
 * Usage: ./masterWorker
 * Exercise:
 * - Compile and run as is.
 * - Uncomment #pragma directive, re-compile and re-run
 * - Compare and trace the different executions.
 */

#include <stdio.h>
#include <omp.h>

int main(int argc, char** argv) {
    int id = -1, numThreads = -1;

//    #pragma omp parallel
    {
        id = omp_get_thread_num();
        numThreads = omp_get_num_threads();

        if ( id == 0 ) {  // thread with ID 0 is master
            printf("Greetings from the master, # %d of %d threads\n",
                            id, numThreads);
        } else {          // threads with IDs > 0 are workers
            printf("Greetings from a worker, # %d of %d threads\n",
                            id, numThreads);
        }
    }

    return 0;
}
```

2.2.6 4. Shared Data Decomposition Pattern: blocking of threads in a parallel for loop

```
/* parallelForBlocks.c
 * ... illustrates the use of OpenMP's default parallel for loop
 *          in which threads iterate through blocks of the index range
 *        (cache-beneficial when accessing adjacent memory locations).
 *
 * Joel Adams, Calvin College, at November 2009.
 * Usage: ./parallelForBlocks [numThreads]
 * Exercise
 */

#include <stdio.h>
```

```
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {
    int rank, i;

    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

    #pragma omp parallel for private(rank, i)
    for (i = 0; i < 8; i++) {
        rank  = omp_get_thread_num();
        printf("Thread %d performed iteration %d\n",
                 rank, i);
    }

    return 0;
}
```

2.2.7 5. Shared Data Decomposition Pattern: striping of threads in a parallel for loop

```
/* parallelForStripes.c
 * ... illustrates how to make OpenMP map threads to
 *        parallel for-loop iterations in 'stripes' instead of blocks
 *        (use only when not accesssing memory).
 *
 * Joel Adams, Calvin College, at November 2009.
 *
 * Usage: ./parallelForStripes [numThreads]
 */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {

    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

    #pragma omp parallel
    {
        int rank  = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
        int i;
        for (i = rank; i < 8; i += numThreads) {
            printf("Thread %d performed iteration %d\n",
                     rank, i);
        }
    }

    return 0;
}
```

2.2.8 6. Collective Communication: Reduction

Once processes have performed independent concurrent computations, possibly on some portion of decomposed data, it is quite commen to then *reduce* those individual computations into one value. In this example, an array of randomly assigned integers represents a set of shared data. All values in the array are summed together by using the OpenMP parallel for pragma with the *reduction(+:sum)* clause on the variable **sum**, which is computed in line 61.

```
1  /* reduction.c
2   * ... illustrates the OpenMP parallel-for loop's reduction clause
3   *
4   * Joel Adams, Calvin College, at November 2009.
5   *
6   * Usage: ./reduction
7   * Exercise:
8   * - Compile and run.  Note that correct output is produced.
9   * - Uncomment #pragma in function parallelSum()
10  *    but leave its reduction clause commented out
11  * - Recompile and rerun.  Note that correct output is NOT produced.
12  * - Uncomment 'reduction(+:sum)' clause of #pragma in parallelSum()
13  * - Recompile and rerun.  Note that correct output is produced again.
14  */
15
16 #include <stdio.h>
17 #include <omp.h>
18 #include <stdlib.h>
19 #include <assert.h>
20
21 void initialize(int* a, int n);
22 int sequentialSum(int* a, int n);
23 int parallelSum(int* a, int n);
24
25 #define SIZE 1000000
26
27 int main(int argc, char** argv) {
28    int array[SIZE];
29
30    initialize(array, SIZE);
31    assert( sequentialSum(array, SIZE) == parallelSum(array, SIZE) );
32    printf("Sequential and parallel functions produced the same result\n");
33
34    return 0;
35 }
36
37 // fill array with random values
38 void initialize(int* a, int n) {
39    int i;
40    for (i = 0; i < n; i++) {
41       a[i] = rand() % 1000;
42    }
43 }
44
45 // sum the array sequentially
46 int sequentialSum(int* a, int n) {
47    int sum = 0;
48    int i;
49    for (i = 0; i < n; i++) {
50       sum += a[i];
51    }
```

```
52     return sum;
53  }
54
55  // sum the array using multiple threads
56  int parallelSum(int* a, int n) {
57     int sum = 0;
58     int i;
59  //    #pragma omp parallel for // reduction(+:sum)
60     for (i = 0; i < n; i++) {
61        sum += a[i];
62     }
63     return sum;
64  }
```

2.2.9 7. Shared Data Pattern: Parallel-for-loop needs non-shared, private variables

```
/* private.c
 * ... illustrates why private variables are needed with OpenMP's parallel for loop
 *
 * Joel Adams, Calvin College, at November 2009.
 * Usage: ./private
 * Exercise:
 * - Run, noting that the serial program produces correct results
 * - Uncomment line A, recompile/run and compare
 * - Recomment line A, uncomment line B, recompile/run and compare
 */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

#define SIZE 100

int main(int argc, char** argv) {
    int i, j, ok = 1;
    int m[SIZE][SIZE];

    // set all array entries to 1
//    #pragma omp parallel for                    // A
//    #pragma omp parallel for private(j)         // B
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            m[i][j] = 1;
        }
    }

    // test (without using threads)
    for (i = 0; i < SIZE; i++) {
        for (j = 0; j < SIZE; j++) {
            if ( m[i][j] != 1 ) {
                printf("Element [%d,%d] not set... \n", i, j);
                ok = 0;
            }
        }
    }

    if ( ok ) {
```

```
        printf("\nAll elements correctly set to 1\n\n");
    }

    return 0;
}
```

2.2.10 8. Race Condition: missing the mutual exclusion patterm

```
/* atomic.c
 * ... illustrates a race condition when multiple threads write to a shared variable
 *  (and explores OpenMP private variables and atomic operations).
 *
 * Joel Adams (Calvin College), November 2009.
 * Usage: ./atomic
 * Exercise:
 *  - Compile and run 10 times; note that it always produces the final balance 0
 *  - To parallelize, uncomment A1+A2, recompile and rerun, compare results
 *  - Try 1: recomment A1+A2, uncomment B1+B2, recompile/run, compare
 *  - To fix: recomment B1+B2, uncomment A1+A2, C1+C2, recompile and rerun, compare
 */

#include<stdio.h>
#include<omp.h>

int main() {
    const int REPS = 1000000;
    int i;
    double balance = 0.0;

    printf("\nYour starting bank account balance is %0.2f\n", balance);

    // simulate many deposits
//    #pragma omp parallel for                      // A1
//    #pragma omp parallel for private(balance)     // B1
    for (i = 0; i < REPS; i++) {
//        #pragma omp atomic                        // C1
        balance += 10.0;
    }

    printf("\nAfter %d $10 deposits, your balance is %0.2f\n",
                REPS, balance);

    // simulate the same number of withdrawals
//    #pragma omp parallel for                      // A2
//    #pragma omp parallel for private(balance)     // B2
    for (i = 0; i < REPS; i++) {
//        #pragma omp atomic                          // C2
        balance -= 10.0;
    }

    // balance should be zero
    printf("\nAfter %d $10 withdrawals, your balance is %0.2f\n\n",
            REPS, balance);

    return 0;
}
```

2.2.11 9. Mutual Exclusion: two ways to ensure

```
/* critical.c
 * ... fixes a race condition when multiple threads write to a shared variable
 *          using the OpenMP critical directive.
 *
 * Joel Adams (Calvin College), November 2009.
 * Usage: ./critical
 * Exercise:
 *  - Compile and run several times; note that it always produces the final balance 0
 *  - Comment out A1+A2; recompile/run and note incorrect result
 *  - To fix: uncomment B1a+B1b+B1c, B2a+B2b+B2c, recompile and rerun, compare
 */

#include<stdio.h>
#include<omp.h>

int main() {
    const int REPS = 1000000;
    int i;
    double balance = 0.0;

    printf("\nYour starting bank account balance is %0.2f\n", balance);

    // simulate many deposits
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
        #pragma omp atomic                          // A1
//        #pragma omp critical                      // B1a
//        {                                         // B1b
        balance += 10.0;
//        }                                         // B1c
    }

    printf("\nAfter %d $10 deposits, your balance is %0.2f\n",
                REPS, balance);

    // simulate the same number of withdrawals
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
        #pragma omp atomic                          // A2
//        #pragma omp critical                      // B2a
//        {                                         // B2b
        balance -= 10.0;
//        }                                         // B2c
  }

    // balance should be zero
    printf("\nAfter %d $10 withdrawals, your balance is %0.2f\n\n",
                REPS, balance);

    return 0;
}
```

2.2.12 10. Mutual Exclusion Pattern: compare performance

```
/* critical.c
 * ... compares the performance of OpenMP's critical and atomic directives
 *
 * Joel Adams (Calvin College), November 2009.
 * Usage: ./critical
 * Exercise:
 *  - Compile, run, compare times for critical vs. atomic
 *  - Compute how much more costly critical is than atomic, on average
 *  - Create an expression setting a shared variable that atomic cannot handle (research)
 */

#include<stdio.h>
#include<omp.h>

void print(char* label, int reps, double balance, double total, double average) {
    printf("\nAfter %d $10 deposits using '%s': \
            \n\t- balance = %0.2f, \
            \n\t- total time = %0.12f, \
            \n\t- average time per deposit = %0.12f\n\n",
               reps, label, balance, total, average);
}

int main() {
    const int REPS = 1000000;
    int i;
    double balance = 0.0,
           startTime = 0.0,
           stopTime = 0.0,
           totalTime = 0.0;

    printf("\nYour starting bank account balance is %0.2f\n", balance);

    // simulate many deposits using atomic
    startTime = omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
        #pragma omp atomic
        balance += 10.0;
    }
    stopTime = omp_get_wtime();
    totalTime = stopTime - startTime;
    print("atomic", REPS, balance, totalTime, totalTime/REPS);


    // simulate the same number of deposits using critical
    balance = 0;
    startTime = omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
         #pragma omp critical
         {
             balance += 10.0;
         }
    }
    stopTime = omp_get_wtime();
    totalTime = stopTime - startTime;
```

```
    print("critical", REPS, balance, totalTime, totalTime/REPS);

    return 0;
}
```

2.2.13 11. Task Decomposition Pattern using OpenMP section directive

```
/* sections.c
 * ... illustrates the use of OpenMP's parallel section/sections directives...
 * Joel Adams, Calvin College, at November 2009.
 */

#include <stdio.h>
#include <omp.h>
#include <stdlib.h>

int main(int argc, char** argv) {

    printf("\nBefore...\n\n");

    #pragma omp parallel sections num_threads(4)
    {
        #pragma omp section
        {
            printf("Section A performed by thread %d\n",
                    omp_get_thread_num() );
        }
        #pragma omp section
        {
            printf("Section B performed by thread %d\n",
                    omp_get_thread_num() );
        }
        #pragma omp section
        {
            printf("Section C performed by thread %d\n",
                    omp_get_thread_num() );
        }
        #pragma omp section
        {
                printf("Section D performed by thread %d\n",
                         omp_get_thread_num() );
        }
    }

    printf("\nAfter...\n\n");

    return 0;
}
```