Parallel Patternlets

CSinParallel Project

February 28, 2016

CONTENTS

1 Parallel Programming Patterns **2**
- 1.1 An organization of parallel patterns . . . 2

2 Message Passing Parallel Patternlets **4**
- 2.1 Source Code . . . 4
- 2.2 00. Single Program, Multiple Data . . . 4
- 2.3 1. The Barrier Coordination Pattern . . . 5
- 2.4 2. The Master-Worker Implementation Strategy Pattern . . . 6
- 2.5 3. Message passing 1, using Send-Receive of a single value . . . 6
- 2.6 4. Message passing 2, using Send-Receive of an array of values . . . 7
- 2.7 5. Message passing 3, using Send-Receive with master-worker pattern . . . 9
- 2.8 6 (text). Data Decomposition: on *equal-sized chunks* using parallel-for . . . 10
- 2.9 6 (visual). Data Decomposition: on *equal-sized chunks* using parallel-for . . . 11
- 2.10 7 (text). Data Decomposition: on *chunks of size 1* using parallel-for . . . 13
- 2.11 7 (visual). Data Decomposition: on *chunks of size 1* using parallel-for . . . 13
- 2.12 8. Broadcast: a special form of message passing . . . 15
- 2.13 9. Collective Communication: Reduction . . . 16
- 2.14 10. Collective Communication: Reduction . . . 17
- 2.15 11. Collective communication: Scatter for message-passing data decomposition . . . 18
- 2.16 12. Collective communication: Gather for message-passing data decomposition . . . 19

3 Shared Memory Parallel Patternlets in OpenMP **22**
- 3.1 Source Code . . . 22
- 3.2 Patternlets Grouped By Type . . . 23

This document contains simple examples of basic elements that are combined to form patterns often used in programs employing parallelism. We call these examples *patternlets* because they are deliberately trivial, small, yet functioning programs that illustrate a basic shell of how a particular parallel pattern is created in a program. They are starting points you can use to create realistic working programs of your own that use the patterns. Before diving into the examples, first there will be some background on parallel programming patterns.

CHAPTER

ONE

PARALLEL PROGRAMMING PATTERNS

Like all programs, parallel programs contain many **patterns**: useful ways of writing code that are used repeatedly by most developers because they work well in practice. These patterns have been documented by developers over time so that useful ways of organizing and writing good parallel code can be learned by new programmers (and even seasoned veterans).

1.1 An organization of parallel patterns

When writing parallel programs, developers use patterns that can be grouped into two main categories:

1. Strategies
2. Concurrent Execution Mechanisms

1.1.1 Strategies

When you set out to write a program, whether it is parallel or not, you should be considering two primary strategic considerations:

1. What *algorithmic strategies* to use
2. Given the algorithmic strategies, what *implementation strategies* to use

In the examples in this document we introduce some well-used patterns for both algorithmic strategies and implementation strategies. Parallel algorithmic strategies primarily have to do with making choices about what tasks can be done concurrently by multiple processing units executing concurrently. Parallel programs often make use of several patterns of implementation strategies. Some of these patterns contribute to the overall structure of the program, and others are concerned with how the data that is being computed by multiple processing units is structured. As you will see, the patternlets introduce more algorithmic strategy patterns and program structure implementation strategy patterns than data structure implementation strategy patterns.

1.1.2 Concurrent Execution Mechanisms

There are a number of parallel code patterns that are closely related to the system or hardware that a program is being written for and the software library used to enable parallelism, or concurrent execution. These *concurrent execution* patterns fall into two major categories:

1. *Process/Thread control* patterns, which dictate how the processing units of parallel execution on the hardware (either a process or a thread, depending on the hardware and software used) are controlled at run time. For the patternlets described in this document, the software libraries that provide system parallelism have these patterns built into them, so they will be hidden from the programmer.

2. *Coordination* patterns, which set up how multiple concurrently running tasks on processing units coordinate to complete the parallel computation desired.

In parallel processing, most software uses one of two major *coordination patterns*:

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

These two types of computation are often realized using two very popular C/C++ libraries:

1. MPI, or Message Passing Interface, for message passing.
2. OpenMP for threaded, shared memory applications.

OpenMP is built on a lower-level POSIX library called Pthreads, which can also be used by itself on shared memory systems.

A third emerging type of parallel implementation involves a *hybrid computation* that uses both of the above patterns together, using a cluster of computers, each of which executes multiple threads. This type of hybrid program often uses MPI and OpenMP together in one program, which runs on multiple computers in a cluster.

This document is split into chapters of examples. There are examples for message passing using MPI and shared memory using OpenMP. (In the future we will include shared memory examples using Pthreads, and hybrid computations using a combination of MPI and OpenMP.)

Most of the examples are illustrated with the C programming language, using standard popular available libraries. In a few cases, C++ is used to illustrate a particular difference in code execution between the two languages or to make use of a C++ BigInt class.

There are many small examples that serve to illustrate a common pattern. They are designed for you to try compiling and running on your own to see how they work. For each example, there are comments within the code to guide you as you try them out. In many cases, there may be code snippets that you can comment and/or uncomment to see how the execution of the code changes after you do so and re-compile it.

Depending on you interest, you can now explore MPI Patternlets or OpenMP Patternlets.

Message Passing Parallel Patternlets

Shared Memory Parallel Patternlets in OpenMP

CHAPTER

TWO

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 multiprocessor machine or a cluster of machines.

2.1 Source Code

Please download all examples from this tarball: `MPI.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.

2.2 00. Single Program, Multiple Data

First let us illustrate the basic components of an MPI program, which by its nature uses a single program that runs on each process. Note what gets printed is different for each process, thus the processes using this one single program can have different data values for its variables. This is why we call it single program, multiple data.

```
/* spmd.c
 * ... illustrates the single program multiple data
 *      (SPMD) pattern using basic MPI commands.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np 4 ./spmd
 *
 * Exercise:
 * - Compile and run.
 * - Compare source code to output.
 * - Rerun, using varying numbers of processes
 *    (i.e., vary the argument to 'mpirun -np').
 * - Explain what "multiple data" values this
 *    "single program" is generating.
 */

#include <stdio.h>   // printf()
```

```
#include <mpi.h>     // MPI functions

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;
}
```

file: patternlets/MPI/00.spmd/spmd.c

2.3 1. The Barrier Coordination Pattern

```
/* barrier.c
 * ... illustrates the behavior of MPI_Barrier() ...
 * Joel Adams, Calvin College, May 2013.
 *
 * Usage: mpirun -np 8 ./barrier
 *
 * Exercise:
 *  - Compile; then run the program several times,
 *    noting the intermixed outputs
 *  - Uncomment the MPI_Barrier() call; then recompile and rerun,
 *    noting how the output changes.
 *  - Explain what effect MPI_Barrier() has on process behavior.
 */

#include <stdio.h>   // printf()
#include <mpi.h>     // MPI

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("Process #%d of %d on %s is BEFORE the barrier.\n",
             id, numProcesses, myHostName);

//    MPI_Barrier(MPI_COMM_WORLD);

    printf("Process #%d of %d on %s is AFTER the barrier.\n",
             id, numProcesses, myHostName);
```

```
    MPI_Finalize();
    return 0;
}
```

file: patternlets/MPI/01.barrier/masterWorker.c

2.4 2. The Master-Worker Implementation Strategy Pattern

```
/* masterWorker.c
 * ... illustrates the basic master-worker pattern in MPI ...
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./masterWorker
 *
 * Exercise:
 * - Compile and run the program, varying N from 1 through 8.
 * - Explain what stays the same and what changes as the
 *    number of processes changes.
 */

#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;
}
```

file: patternlets/MPI/02.masterWorker/masterWorker.c

2.5 3. Message passing 1, using Send-Receive of a single value

```
/* messagePassing.c
 * ... illustrates the use of the MPI_Send() and MPI_Recv() commands...
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./messagePassing
 *
```

```
 * Exercise:
 * - Compile and run, using N = 4, 6, 8, and 10 processes.
 * - Use source code to trace execution.
 * - Explain what each process:
 * -- computes
 * -- sends
 * -- receives
 * -- outputs.
 */

#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;
}
```

file: patternlets/MPI/03.messagePassing/messagePassing.c

2.6 4. Message passing 2, using Send-Receive of an array of values

```
/* messagePassing2.c
 * ... illustrates using MPI_Send() and MPI_Recv() commands on arrays...
 * While this example sends and receives char arrays (strings),
 *  the same approach works on arrays of numbers or other types.
 * Joel Adams, Calvin College, September 2013.
```

```
 *
 * Usage: mpirun -np N ./messagePassing2
 *
 * Exercise:
 * - Compile and run, varying N: 1, 2, 4, 8.
 * - Trace execution using source code.
 * - Compare to messagePassing1.c; note send/receive differences.
 */

#include <stdio.h>   // printf()
#include <mpi.h>     // MPI
#include <stdlib.h>  // malloc()
#include <string.h>  // strlen()

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

int main(int argc, char** argv) {
    int id = -1, numProcesses = -1, length = -1;
    char * sendString = NULL;
    char * receivedString = NULL;
    char hostName[MPI_MAX_PROCESSOR_NAME];
    MPI_Status status;
    const int SIZE = (32+MPI_MAX_PROCESSOR_NAME) * sizeof(char);

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

    if (numProcesses > 1 && !odd(numProcesses) ) {
        sendString = (char*) malloc( SIZE );
        receivedString = (char*) malloc( SIZE );
        sprintf(sendString, "Process %d is on host \"%s\"", id, hostName);

        if ( odd(id) ) {  // odd processes send, then receive
            MPI_Send(sendString, strlen(sendString)+1,
                     MPI_CHAR, id-1, 1, MPI_COMM_WORLD);
            MPI_Recv(receivedString, SIZE, MPI_CHAR, id-1, 2,
                     MPI_COMM_WORLD, &status);
        } else {          // even processes receive, then send
            MPI_Recv(receivedString, SIZE, MPI_CHAR, id+1, 1,
                     MPI_COMM_WORLD, &status);
            MPI_Send(sendString, strlen(sendString)+1,
                     MPI_CHAR, id+1, 2, MPI_COMM_WORLD);
        }

        printf("\nProcess %d of %d received the message:\n\t'%s'\n",
                id, numProcesses, receivedString);

        free(sendString);
        free(receivedString);
    } 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;
}
```

file: patternlets/MPI/04.messagePassing2/messagePassing2.c

2.7 5. Message passing 3, using Send-Receive with master-worker pattern

```
/* messagePassing3.c
 * ... illustrates the use of MPI_Send() and MPI_Recv(),
 *      in combination with the master-worker pattern.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./messagePassing3
 *
 * Exercise:
 * - Run the program, varying the value of N from 1-8.
 * - Explain the behavior you observe.
 */

#include <stdio.h>    // printf()
#include <string.h>   // strlen()
#include <mpi.h>      // MPI

#define MAX 256

int main(int argc, char** argv) {
    int id = -1, numProcesses = -1;
    char sendBuffer[MAX] = {'\0'};
    char recvBuffer[MAX] = {'\0'};
    MPI_Status status;

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

    if (numProcesses > 1) {
        if ( id == 0 ) {                              // master:
            sprintf(sendBuffer, "%d", id);            //  create msg

            MPI_Send(sendBuffer,                      //  msg sent
                     strlen(sendBuffer) + 1,          //  num chars + NULL
                     MPI_CHAR,                        //  type
                     id+1,                            //  destination
                     1,                               //  tag
                     MPI_COMM_WORLD);                 //  communicator

            MPI_Recv(recvBuffer,                      //  msg received
                     MAX,                             //  buffer size
                     MPI_CHAR,                        //  type
                     numProcesses-1,                  //  sender
                     1,                               //  tag
                     MPI_COMM_WORLD,                  //  communicator
                     &status);                        //  recv status
        } else {                                      // workers:
            MPI_Recv(recvBuffer,                      //  msg received
                     MAX,                             //  buffer size
                     MPI_CHAR,                        //  type
```

```
                    MPI_ANY_SOURCE,                 //  sender (anyone)
                    1,                              //  tag
                    MPI_COMM_WORLD,                 //  communicator
                    &status);                       //  recv status

            printf("Process #%d of %d received %s\n", // show msg
                    id, numProcesses, recvBuffer);

            // build msg to send by appending id to msg received
            sprintf(sendBuffer, "%s %d", recvBuffer, id);

            MPI_Send(sendBuffer,                    //  msg to send
                     strlen(sendBuffer) + 1,        //  num chars + NULL
                     MPI_CHAR,                      //  type
                     (id+1) % numProcesses,         //  destination
                     1,                             //  tag
                     MPI_COMM_WORLD);               //  communicator
        }
    } else {
        printf("\nPlease run this program with at least 2 processes\n\n");
    }

    MPI_Finalize();
    return 0;
}
```

file: patternlets/MPI/05.messagePassing3/messagePassing3.c

2.8 6 (text). Data Decomposition: on *equal-sized chunks* using parallel-for

In this example, the data being decomposed is simply the set of integers from zero to REPS * numProcesses, which are used in the for loop.

```
/* parallelLoopEqualChunks.c
 * ... illustrates the parallel for loop pattern in MPI
 *       in which processes perform the loop's iterations in equal-sized 'chunks'
 *       (preferable when loop iterations access memory/cache locations) ...
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./parallelForEqualChunks
 *
 * Exercise:
 * - Compile and run, varying N: 1, 2, 3, 4, 5, 6, 7, 8
 * - Change REPS to 16, save, recompile, rerun, varying N again.
 * - Explain how this pattern divides the iterations of the loop
 *    among the processes.
 */

#include <stdio.h> // printf()
#include <mpi.h>   // MPI
#include <math.h>  // ceil()

int main(int argc, char** argv) {
    const int REPS = 8;
    int id = -1, numProcesses = -1, i = -1,
```

```
        start = -1, stop = -1, chunkSize = -1;

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

    chunkSize = (int)ceil(((double)REPS) / numProcesses); // find chunk size
    start = id * chunkSize;                               // find starting index
                                                          // find stopping index:
    if ( id < numProcesses - 1 ) {                        //  if not the last process
        stop = (id + 1) * chunkSize;                      //   stop where next process starts
    } else {                                              //  else
        stop = REPS;                                      //   last process does leftovers
    }

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

    MPI_Finalize();
    return 0;
}
```

file: patternlets/MPI/06.parallelLoop-equalChunks/textual/parallelLoopEqualChunks.c

2.9 6 (visual). Data Decomposition: on *equal-sized chunks* using parallel-for

In this example, we can visually see how the slicing of data used in iterations of a nested for loop is working. Run it to see the effect!

```
/* parallelForBlocks.c is a graphical illustration of
 *  the 'blocks' version of the parallel for loop design pattern,
 *  using Argonne Labs' MPE graphics library for X11 systems.
 *
 * Summer 2013, Joel Adams, Calvin College.
 *
 * Usage: mpirun -np N ./parallelForBlocks
 *        Click the mouse in the window to terminate the program.
 *        You must have an X11 server running.
 *
 * Exercise: Run the program, varying N from 1 - 32,
 *           and compare to the 'Stripes' version...
 */

#include <mpi.h>     // MPI
#include <mpe.h>     // MPE
#include <stdlib.h>  // getenv()
#include <string.h>  // strcmp()
#include <stdio.h>   // printf(), etc.

/*
 * getDisplay() retrieves the DISPLAY environment info
 */

char* getDisplay() {
```

```
        char * display = getenv("DISPLAY");
        if ( strncmp(display, "(null)", 7) == 0 ) {
                fprintf(stderr, "\n*** Fatal: DISPLAY variable not set.\n");
                exit(1);
        }
        return display;
}


int main(int argc, char* argv[]) {
    const int  WINDOW_WIDTH = 800;
    const int  WINDOW_HEIGHT = 400;
    int        x = 0, y = 0;
    int        id = -1, numProcesses = -1;
    int        button = -1;
    int        blockSize = 0;
    int        yStart, yEnd;
    MPE_XGraph canvas;
    MPE_Color *colors = NULL;
    MPE_Color  myColor = 0;

    // initialize environment, variables, etc.
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    MPE_Open_graphics( &canvas, MPI_COMM_WORLD,
                       getDisplay(),
                       5, 5,
                       WINDOW_WIDTH, WINDOW_HEIGHT, 0 );
    colors = malloc( numProcesses * sizeof(MPE_Color) );
    MPE_Make_color_array(canvas, numProcesses, colors);
    myColor = colors[id];
    blockSize = WINDOW_HEIGHT / numProcesses;
    yStart = id * blockSize;
    yEnd = yStart + blockSize;

    // processes draw the window's pixels in stripes
    for (y = yStart; y < yEnd; y++) {
       for (x = 0; x < WINDOW_WIDTH; x++) {
          MPE_Draw_point(canvas, x, y, myColor);
          MPE_Update(canvas);  // update here to slow things down
       }
    }

    // pause until mouse-click so the program doesn't terminate
    if (id == 0) {
       printf("\nClick in the window to continue...\n");
       MPE_Get_mouse_press( canvas, &x, &y, &button );
    }

    // clean up
    MPE_Close_graphics( &canvas);
    MPI_Finalize();
    free(colors);
    if (id == 0) printf("Program complete.\n\n");
    return 0;
}
```

file: patternlets/MPI/06.parallelLoop-equalChunks/visual/parallelForBlocks.c

2.10 7 (text). Data Decomposition: on *chunks of size 1* 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.

```
/* parallelLoopChunksOf1.c
 * ... illustrates the parallel for loop pattern in MPI
 *        in which processes perform the loop's iterations in 'chunks'
 *      of size 1 (simple, and useful when loop iterations
 *      do not access memory/cache locations) ...
 * Note this is much simpler than the 'equal chunks' loop.
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./parallelForSlices
 *
 * Exercise:
 * - Compile and run, varying N: 1, 2, 3, 4, 5, 6, 7, 8
 * - Change REPS to 16, save, recompile, rerun, varying N again.
 * - Explain how this pattern divides the iterations of the loop
 *    among the processes.
 */

#include <stdio.h>  // printf()
#include <mpi.h>    // MPI

int main(int argc, char** argv) {
    const int REPS = 8;
    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 < REPS; i += numProcesses) {
        printf("Process %d is performing iteration %d\n", id, i);
    }

    MPI_Finalize();
    return 0;
}
```

file: patternlets/MPI/07.parallelLoop-chunksOf1/textual/parallelLoopChunksOf1.c

2.11 7 (visual). Data Decomposition: on *chunks of size 1* using parallel-for

In this example you can see how blocks of values within a matrix might be assigned to each process. Run it to see the effect!

```
/* parallelForStripes.c is a graphical illustration of
 *  the 'slicing' version of the parallel for loop design pattern,
 *  using Argonne Labs' MPE graphics library for X11 systems.
 *
 * Summer 2013, Joel Adams, Calvin College.
 *
 * Usage: mpirun -np N ./parallelForStripes
 *        Click the mouse in the window to terminate the program.
 *        You must have an X11 server running.
 *
 * Exercise: Run the program, varying N from 1 - 32,
 *           and compare to the 'Blocks' version...
 */

#include <mpi.h>     // MPI
#include <mpe.h>     // MPE
#include <stdlib.h>  // getenv()
#include <string.h>  // strcmp()
#include <stdio.h>   // printf(), etc.

/*
 * getDisplay() retrieves the DISPLAY environment info
 */

char* getDisplay() {
        char * display = getenv("DISPLAY");
        if ( strncmp(display, "(null)", 7) == 0 ) {
                fprintf(stderr, "\n*** Fatal: DISPLAY variable not set.\n");
                exit(1);
        }
        return display;
}


int main(int argc, char* argv[]) {
    const int  WINDOW_WIDTH = 800;
    const int  WINDOW_HEIGHT = 400;
    int        x = 0, y = 0;
    int        id = -1, numProcesses = -1;
    int        button = -1;
    MPE_XGraph canvas;
    MPE_Color *colors = NULL;
    MPE_Color  myColor = 0;

    // initialize environment, variables, etc.
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    MPI_Comm_size(MPI_COMM_WORLD, &numProcesses);
    MPE_Open_graphics( &canvas, MPI_COMM_WORLD,
                       getDisplay(),
                       5, 5,
                       WINDOW_WIDTH, WINDOW_HEIGHT, 0 );
    colors = malloc( numProcesses * sizeof(MPE_Color) );
    MPE_Make_color_array(canvas, numProcesses, colors);
    myColor = colors[id];

    // each process does a block of the window's pixels
    for (y = id; y < WINDOW_HEIGHT; y += numProcesses) {
```

```
        for (x = 0; x < WINDOW_WIDTH; x++) {
           MPE_Draw_point(canvas, x, y, myColor);
           MPE_Update(canvas);  // update here to slow things down
        }
    }

    // pause until mouse-click so the program doesn't terminate
    if (id == 0) {
       printf("\nClick in the window to continue...\n");
       MPE_Get_mouse_press( canvas, &x, &y, &button );
    }

    // clean up
    MPE_Close_graphics( &canvas);
    MPI_Finalize();
    free(colors);
    if (id == 0) printf("Program complete.\n\n");
    return 0;
}
```

file: atternlets/MPI/07.parallelLoop-chunksOf1/visual/parallelForSlices.c

2.12 8. 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.

```
/* broadcast.c
 * ... illustrates the use of MPI_Bcast()...
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./broadcast
 *
 * Exercise:
 * - Compile and run, using 2, 4, and 8 processes
 * - Use source code to trace execution and output
 * - Explain behavior/effect of MPI_Bcast().
 */

#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;
}
```

file: patternlets/MPI/08.broadcast/broadcast.c

2.13 9. Collective Communication: Reduction

Once processes have performed independent concurrent computations, possibly on some portion of decomposed data, it is quite common 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.

```
/* reduction.c
 * ... illustrates the use of MPI_Reduce()...
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: mpirun -np N ./reduction
 *
 * Exercise:
 * - Compile and run, varying N: 4, 6, 8, 10.
 * - Explain behavior of MPI_Reduce().
 */

#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;
}
```

file: patternlets/MPI/09.reduction/reduction.c

2.14 10. Collective Communication: Reduction

Here is a second reduction example using arrays of data.

```
/* reduction2.c
 * ... illustrates the use of MPI_Reduce() using arrays...
 * Joel Adams, Calvin College, January 2015.
 *
 * Usage: mpirun -np 4 ./reduction2
 *
 * Exercise:
 * - Compile and run, comparing output to source code.
 * - Uncomment the 'commented out' call to printArray.
 * - Save, recompile, rerun, comparing output to source code.
 * - Explain behavior of MPI_Reduce() in terms of
 *     srcArr and destArr.
 */

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

#define ARRAY_SIZE 5

void printArray(int id, char* arrayName, int* array, int SIZE);

int main(int argc, char** argv) {
    int myRank = -1;
    int srcArr[ARRAY_SIZE] = {0};
    int destArr[ARRAY_SIZE] = {0};

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

    if (myRank == 0) {
        printf("\nBefore reduction: ");
        printArray(myRank, "destArr", destArr, ARRAY_SIZE);
```

```
    }

    for (unsigned i = 0; i < ARRAY_SIZE; i++) {
        srcArr[i] = myRank * i;
    }

//    printArray(myRank, "srcArr", srcArr, ARRAY_SIZE);

    MPI_Reduce(srcArr, destArr, ARRAY_SIZE, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);

    if (myRank == 0) {
        printf("\nAfter reduction:  ");
        printArray(myRank, "destArr", destArr, ARRAY_SIZE);
        printf("\n");
    }

    MPI_Finalize();

    return 0;
}

/* utility to display an array
 * params: id, the rank of the current process
 *         arrayName, the name of the array being displayed
 *         array, the array being displayed
 *         SIZE, the number of items in array.
 * postcondition:
 *         the id, name, and items in array have been printed to stdout.
 */
void printArray(int id, char* arrayName, int * array, int SIZE) {
    printf("Process %d, %s: [", id, arrayName);
    for (int i = 0; i < SIZE; i++) {
        printf("%3d", array[i]);
        if (i < SIZE-1) printf(",");
    }
    printf("]\n");
}
```

file: patternlets/MPI/10.reduction2/reduction2.c

2.15 11. 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 pattern 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, November 2009.
 *
 * Usage: mpirun -np N ./scatter
 *
 * Exercise:
 * - Compile and run, varying N: 1, 2, 4, 8
 * - Trace execution through source code.
 * - Explain behavior/effect of MPI_Scatter().
```

```
 */

#include <mpi.h>      // MPI
#include <stdio.h>    // printf(), etc.
#include <stdlib.h>   // malloc()

void print(int id, char* arrName, int* arr, int arrSize);

int main(int argc, char** argv) {
    const int MAX = 8;
    int* arrSend = NULL;
    int* arrRcv = NULL;
    int numProcs = -1, myRank = -1, numSent = -1;

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

    if (myRank == 0) {                                       // master process:
        arrSend = (int*) malloc( MAX * sizeof(int) );        //  allocate array1
        for (int i = 0; i < MAX; i++) {                      //  load with values
            arrSend[i] = (i+1) * 11;
        }
        print(myRank, "arrSend", arrSend, MAX);              //  display array1
    }

    numSent = MAX / numProcs;                                // all processes:
    arrRcv = (int*) malloc( numSent * sizeof(int) );         //  allocate array2

    MPI_Scatter(arrSend, numSent, MPI_INT, arrRcv,           //  scatter array1
                numSent, MPI_INT, 0, MPI_COMM_WORLD); //   into array2

    print(myRank, "arrRcv", arrRcv, numSent);                // display array2

    free(arrSend);                                           // clean up
    free(arrRcv);
    MPI_Finalize();
    return 0;
}

void print(int id, char* arrName, int* arr, int arrSize) {
    printf("Process %d, %s: ", id, arrName);
    for (int i = 0; i < arrSize; i++) {
        printf(" %d", arr[i]);
    }
    printf("\n");
}
```

file: patternlets/MPI/11.scatter/scatter.c

2.16 12. 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 pattern 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.
 *
 * Usage: mpirun -np N ./gather
 *
 * Exercise:
 * - Compile and run, varying N: 1, 2, 4, 8.
 * - Trace execution through source.
 * - Explain behavior of MPI_Gather().
 */

#include <mpi.h>       // MPI
#include <stdio.h>     // printf()
#include <stdlib.h>    // malloc()

void print(int id, char* arrName, int* arr, int arrSize);

#define SIZE 3

int main(int argc, char** argv) {
   int  computeArray[SIZE];                          // array1
   int* gatherArray = NULL;                          // array2
   int  numProcs = -1, myRank = -1,
        totalGatheredVals = -1;

   MPI_Init(&argc, &argv);                           // initialize
   MPI_Comm_size(MPI_COMM_WORLD, &numProcs);
   MPI_Comm_rank(MPI_COMM_WORLD, &myRank);
                                                     // all processes:
   for (int i = 0; i < SIZE; i++) {                  //  load array1 with
      computeArray[i] = myRank * 10 + i;             //   3 distinct values
   }

   print(myRank, "computeArray", computeArray,       //  show array1
         SIZE);

   if (myRank == 0) {                                // master:
      totalGatheredVals = SIZE * numProcs;           //  allocate array2
      gatherArray = (int*) malloc( totalGatheredVals * sizeof(int) );
   }

   MPI_Gather(computeArray, SIZE, MPI_INT,           //  gather array1 vals
              gatherArray, SIZE, MPI_INT,            //   into array2
              0, MPI_COMM_WORLD);                    //   at master process

   if (myRank == 0) {                                // master process:
      print(myRank, "gatherArray",                   //  show array2
            gatherArray, totalGatheredVals);
   }

   free(gatherArray);                                // clean up
   MPI_Finalize();
   return 0;
}

void print(int id, char* arrName, int* arr, int arrSize) {
```

```
    printf("Process %d, %s: ", id, arrName);
    for (int i = 0; i < arrSize; i++) {
        printf(" %d", arr[i]);
    }
    printf("\n");
}
```

file: patternlets/MPI/12.gather/gather.c

CHAPTER

THREE

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 model 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. OpenMP is a standard that compilers who implement it must adhere to.

The following are examples of C code with OpenMP pragmas. There is one C++ example that is used to illustrate a point about that language. The first three are basic illustrations so you can get used to the OpenMP pragmas and conceptualize the two primary patterns used as **program structure implementation strategies** that almost all shared-memory parallel programs have:

- **fork/join**: forking threads and joining them back, and
- **single program, multiple data**: writing one program in which separate threads maybe performing different computations simultaneously on different data, some of which might be shared in memory.

The rest of the examples illustrate how to implement other patterns along with the above two and what can go wrong when mutual exclusion is not properly ensured.

Note: by default OpenMP uses the **Thread Pool** pattern of concurrent execution control. OpenMP programs initialize 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. In addition, the **multiple instruction, multiple data** pattern is used in OpenMP programs because multiple threads can be executing different instructions on different data in memory at the same point in time.

3.1 Source Code

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

A C code file and a Makefile for each example below can be found in subdirectories of the openMP directory created by extracting the above tarball. The number for each example below corresponds to one used in subdirectory names containing each one.

To compile and run these examples, you will need a C compiler with OpenMP. The GNU C compiler is OpenMP compliant. We assume you are building and executing these on a Unix command line.

3.2 Patternlets Grouped By Type

If you are working on these for the first time, you may want to visit them in order. If you are returning to review a particular patternlet or the pattern categorization diagram, you can refer to them individually.

Shared Memory Program Structure and Coordination Patterns

Data Decomposition Algorithm Strategies and Related Coordination Strategies

Patterns used when threads share data values

Task Decomposition Algorithm Strategies

Categorizing Patterns

3.2.1 Shared Memory Program Structure and Coordination Patterns

0. Program Structure Implementation Strategy: The basic fork-join pattern

file: openMP/00.forkJoin/forkJoin.c

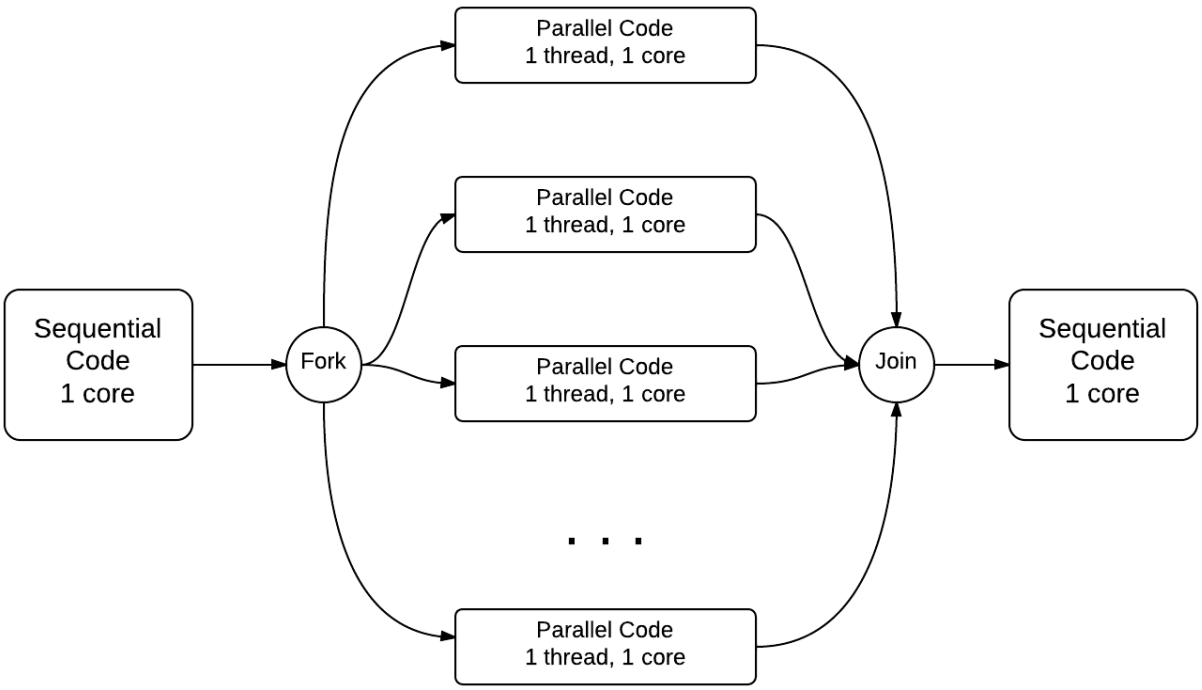
Build inside 00.forkJoin directory:

```
make forkjoin
```

Execute on the command line inside 00.forkJoin directory:

```
./forkjoin
```

The *omp parallel* pragma on line 21, when uncommented, tells the compiler to fork a set of threads to execute the next line of code (later you will see how this is done for a block of code). You can conceptualize how this works using the following diagram, where time is moving from left to right:



Observe what happens on the machine where you are running this code, both when you have the pragma commented (no fork) and when you uncomment it (adding a fork).

Note that in OpenMP the join is implicit and does not require a pragma directive.

```
/* forkJoin.c
 * ... illustrates the fork-join pattern
 *      using OpenMP's parallel directive.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./forkJoin
 *
 * Exercise:
 * - Compile & run, uncomment the pragma,
 *    recompile & run, compare results.
 */

#include <stdio.h>     // printf()
#include <omp.h>       // OpenMP

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

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

//    #pragma omp parallel
    printf("\nDuring...");

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

    return 0;
}
```

1. Program Structure Implementation Strategy: Fork-join with setting the number of threads

file openMP/01.forkJoin2/forkJoin2.c

Build inside 01.forkJoin2 directory:

```
make forkjoin2
```

Execute on the command line inside 01.forkJoin2 directory:

```
./forkjoin2
```

This code illustrates that one program can fork and join more than once and that programmers can set the number of threads to use in the parallel forked code.

Note on line 28 there is an OpenMP function called *omp_set_num_threads* for setting the number of threads to use for each *fork*, which occur when the omp_parallel pragma is used. Also note on line 35 that you can set the number of threads for the very next fork indicated by an omp_parallel pragma by augmenting the pragma as shown in line 35. Follow the instructions in the header of the code file to understand the difference between these.

```
/* forkJoin2.c
 * ... illustrates the fork-join pattern
 *      using multiple OpenMP parallel directives,
 *      and changing the number of threads two ways.
 *
 * Joel Adams, Calvin College, May 2013.
```

```
 *
 * Usage: ./forkJoin2
 *
 * Exercise:
 * - Compile & run, compare results to source.
 * - Predict how many threads will be used in 'Part IV'?
 * - Uncomment 'Part IV', recompile, rerun.
 */

#include <stdio.h>    // printf()
#include <omp.h>      // OpenMP

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

    printf("\nBeginning\n");

    #pragma omp parallel
    printf("\nPart I");

    printf("\n\nBetween I and II...\n");

    omp_set_num_threads(3);

    #pragma omp parallel
    printf("\nPart II...");

    printf("\n\nBetween II and III...\n");

    #pragma omp parallel num_threads(5)
    printf("\nPart III...");
/*
    printf("\n\nBetween III and IV...\n");

    #pragma omp parallel
    printf("\nPart IV...");
*/
    printf("\n\nEnd\n\n");

    return 0;
}
```

2. Program Structure Implementation Strategy: Single Program, multiple data

file: openMP/02.spmd/spmd.c

Build inside 02.spmd directory:

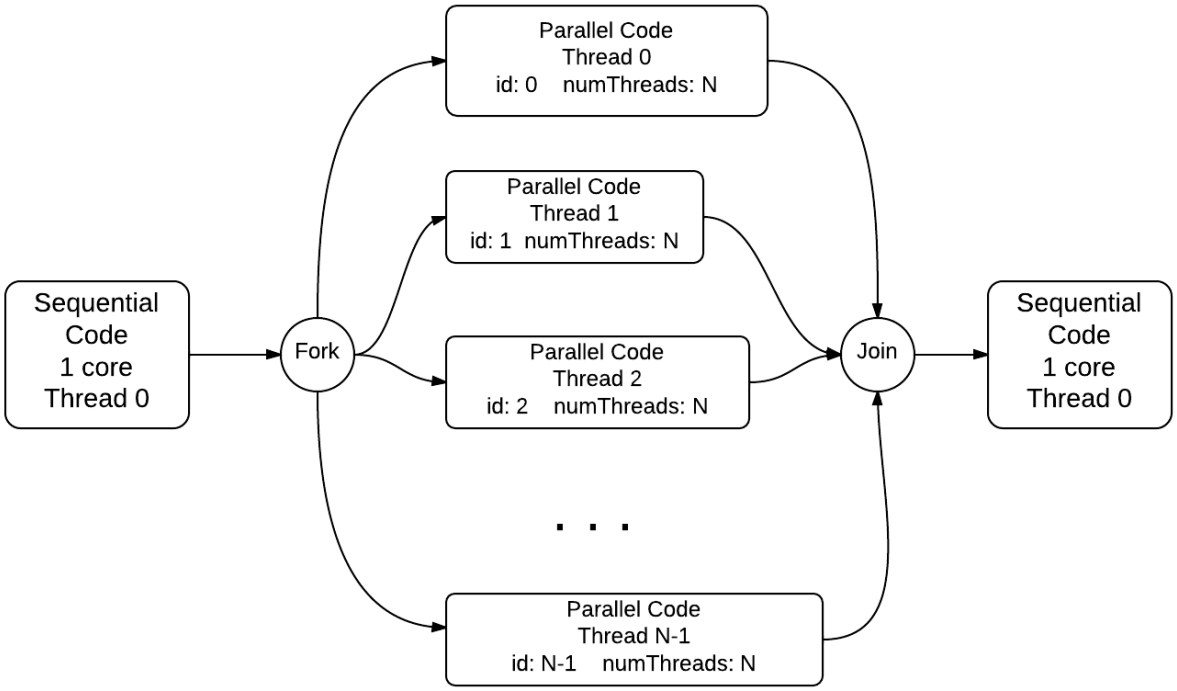
```
make spmd
```

Execute on the command line inside 02.spmd directory:

```
./spmd
```

Note how there are OpenMP functions to obtain a thread number and the total number of threads. We have one program, but multiple threads executing in the forked section, each with a copy of the id and num_threads variables. Programmers write one program, but write it in such a way that each thread has its own data values for particular variables. This is why this is called the *single program, multiple data* (SPMD) pattern.

Most parallel programs use this SPMD pattern, because writing one program is ultimately the most efficient method for programmers. It does require you as a programmer to understand how this works, however. Think carefully about how each thread executing in parallel has its own set of variables. Conceptually, it looks like this, where each thread has its own memory for the variables id and numThreads:



When the pragma is uncommented in the code below, note what the default number of threads is. Here the threads are forked and execute the block of code inside the curly braces on lines 22 through 26. This is how we can have a block of code executed concurrently on each thread.

When you execute the parallel version containing the pragma (uncommenting line 20), what do you observe about the order of the printed lines? Run the program multiple times– does the ordering change? This illustrates an important point about threaded programs: *the ordering of execution of statements between threads is not guaranteed.* This is also illustrated in the diagram above.

```
/* spmd.c
 * ... illustrates the single-program-multiple-data (SPMD)
 *      pattern using two basic OpenMP commands...
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./spmd
 *
 * Exercise:
 * - Compile & run
 * - Uncomment pragma, recompile & run, compare results
 */

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

int main(int argc, char** argv) {
    printf("\n");

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

    printf("\n");
    return 0;
}
```

3. Program Structure Implementation Strategy: Single Program, multiple data with user-defined number of threads

file: openMP/03.spmd2/spmd2.c

Build inside 03.spmd2 directory:

```
make spmd2
```

Execute on the command line inside 03.spmd2 directory:

```
./spmd2 4
Replace 4 with other values for the number of threads
```

Here we enter the number of threads to use on the command line. This is a useful way to make your code versatile so that you can use as many threads as you would like.

```
/* spmd2.c
 * ... illustrates the SPMD pattern in OpenMP,
 *         using the commandline arguments
 *      to control the number of threads.
 *
 * Joel Adams, Calvin College, November 2009.
```

```
 *
 * Usage: ./spmd2 [numThreads]
 *
 * Exercise:
 * - Compile & run with no commandline args
 * - Rerun with different commandline args,
 *    until you see a problem with thread ids
 * - Fix the race condition
 *    (if necessary, compare to 02.spmd)
 */

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

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

    printf("\n");
    if (argc > 1) {
        omp_set_num_threads( atoi(argv[1]) );
    }

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

    printf("\n");
    return 0;
}
```

4. Coordination: Synchronization with a Barrier

file: openMP/04.barrier/barrier.c

Build inside 04.barrier directory:

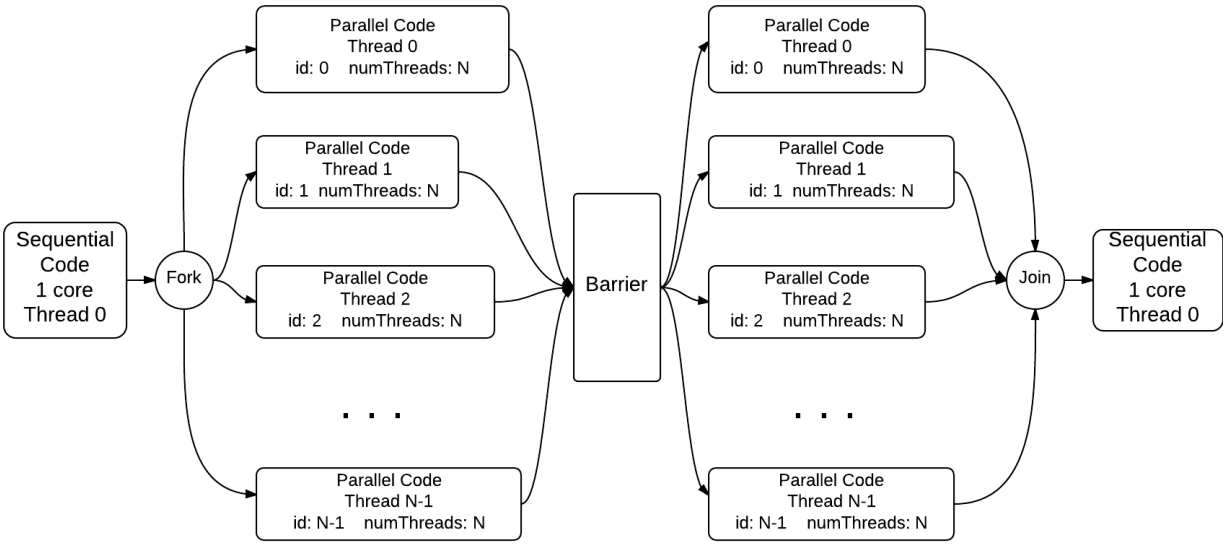
```
make barrier
```

Execute on the command line inside 04.barrier directory:

```
./barrier 4
Replace 4 with other values for the number of threads
```

The barrier pattern is used in parallel programs to ensure that all threads complete a parallel section of code before execution continues. This can be necessary when threads are generating computed data (in an array, for example) that needs to be completed for use in another computation.

Conceptually, the running code is excuting like this:



Note what happens with and without the commented pragma on line 31.

```
1  /* barrier.c
2   * ... illustrates the use of the OpenMP barrier command,
3   *         using the commandline to control the number of threads...
4   *
5   * Joel Adams, Calvin College, May 2013.
6   *
7   * Usage: ./barrier [numThreads]
8   *
9   * Exercise:
10  * - Compile & run several times, noting interleaving of outputs.
11  * - Uncomment the barrier directive, recompile, rerun,
12  *    and note the change in the outputs.
13  */
14
15 #include <stdio.h>
16 #include <omp.h>
17 #include <stdlib.h>
18
19 int main(int argc, char** argv) {
20     printf("\n");
21     if (argc > 1) {
22         omp_set_num_threads( atoi(argv[1]) );
23     }
24
25     #pragma omp parallel
26     {
27         int id = omp_get_thread_num();
28         int numThreads = omp_get_num_threads();
29         printf("Thread %d of %d is BEFORE the barrier.\n", id, numThreads);
30
31 //        #pragma omp barrier
32
33         printf("Thread %d of %d is AFTER the barrier.\n", id, numThreads);
34     }
35
36     printf("\n");
```

```
37      return 0;
38  }
```

5. Program Structure: The Master-Worker Implementation Strategy

file: openMP/05.masterWorker/masterWorker.c

Build inside 05.masterWorker directory:

```
make masterWorker
```

Execute on the command line inside 05.masterWorker directory:

```
./masterWorker 4
Replace 4 with other values for the number of threads
```

Once you have mastered the notion of fork-join and single-program, multiple data, the next common pattern that programmers use in association with these patterns is to have one thread, called the master, execute one block of code when it forks while the rest of the threads, called workers, execute a different block of code when they fork. This is illustrated in this simple example (useful code would be more complicated).

```
1   /* masterWorker.c
2    * ... illustrates the master-worker pattern in OpenMP
3    *
4    * Joel Adams, Calvin College, November 2009.
5    *
6    * Usage: ./masterWorker
7    *
8    * Exercise:
9    * - Compile and run as is.
10   * - Uncomment #pragma directive, re-compile and re-run
11   * - Compare and trace the different executions.
12   */
13
14  #include <stdio.h>   // printf()
15  #include <stdlib.h>   // atoi()
16  #include <omp.h>     // OpenMP
17
18  int main(int argc, char** argv) {
19      printf("\n");
20      if (argc > 1) {
21          omp_set_num_threads( atoi(argv[1]) );
22      }
23
24  //    #pragma omp parallel
25      {
26          int id = omp_get_thread_num();
27          int numThreads = omp_get_num_threads();
28
29          if ( id == 0 ) {  // thread with ID 0 is master
30              printf("Greetings from the master, # %d of %d threads\n",
31                                  id, numThreads);
32          } else {          // threads with IDs > 0 are workers
33              printf("Greetings from a worker, # %d of %d threads\n",
34                                  id, numThreads);
35          }
36      }
37
```

```
38      printf("\n");
39
40      return 0;
41  }
```

3.2.2 Data Decomposition Algorithm Strategies and Related Coordination Strategies

6. Shared Data Decomposition Algorithm Strategy: chunks of data per thread using a parallel for loop implementation strategy

file: openMP/06.parallelLoop-equalChunks/parallelLoopEqualChunks.c

Build inside 06.parallelLoop-equalChunks directory:

```
make parallelLoopEqualChunks
```

Execute on the command line inside 06.parallelLoop-equalChunks directory:

```
./parallelLoopEqualChunks 4
Replace 4 with other values for the number of threads, or leave off
```

An iterative for loop is a remarkably common pattern in all programming, primarily used to perform a calculation N times, often over a set of data containing N elements, using each element in turn inside the for loop. If there are no dependencies between the calculations (i.e. the order of them is not important), then the code inside the loop can be split between forked threads. When doing this, a decision the programmer needs to make is to decide how to partition the work between the threads by answering this question:

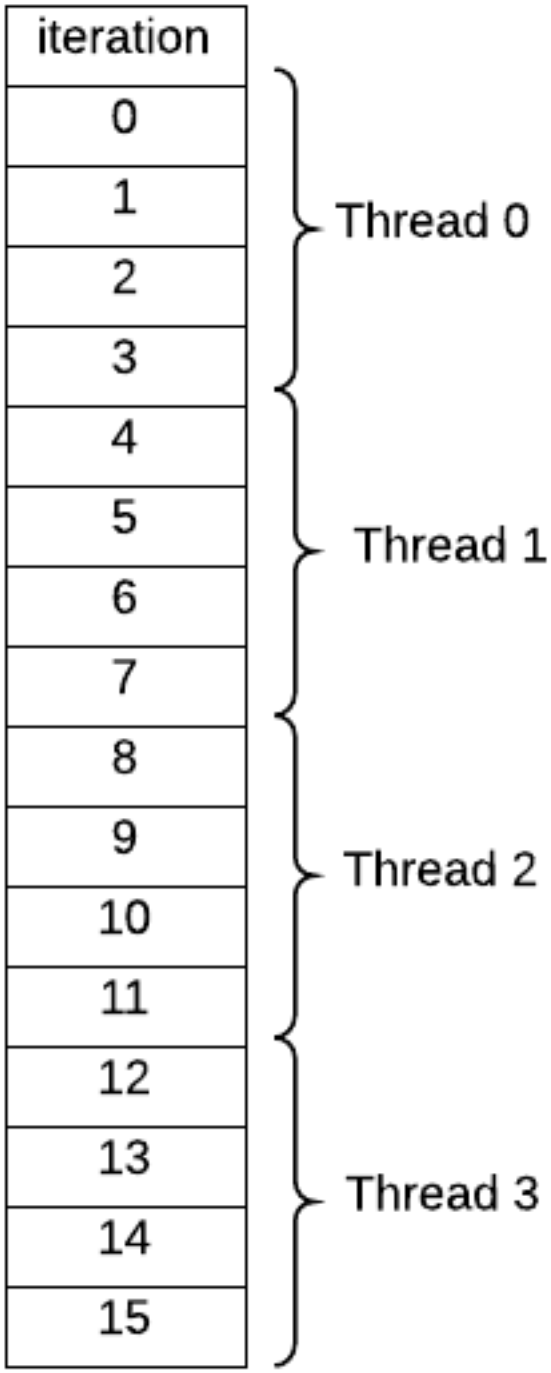
- How many and which iterations of the loop will each thread complete on its own?

We refer to this as the **data decomposition** pattern because we are decomposing the amount of work to be done (typically on a set of data) across multiple threads. In the following code, this is done in OpenMP using the *omp parallel for* pragma just in front of the for statement (line 27) in the following code.

```
1   /* parallelLoopEqualChunks.c
2    * ... illustrates the use of OpenMP's default parallel for loop in which
3    *          threads iterate through equal sized chunks of the index range
4    *        (cache-beneficial when accessing adjacent memory locations).
5    *
6    * Joel Adams, Calvin College, November 2009.
7    *
8    * Usage: ./parallelLoopEqualChunks [numThreads]
9    *
10   * Exercise
11   * - Compile and run, comparing output to source code
12   * - try with different numbers of threads, e.g.: 2, 3, 4, 6, 8
13   */
14
15  #include <stdio.h>    // printf()
16  #include <stdlib.h>   // atoi()
17  #include <omp.h>      // OpenMP
18
19  int main(int argc, char** argv) {
20      const int REPS = 16;
21
22      printf("\n");
23      if (argc > 1) {
```

```
24          omp_set_num_threads( atoi(argv[1]) );
25      }
26
27      #pragma omp parallel for
28      for (int i = 0; i < REPS; i++) {
29          int id = omp_get_thread_num();
30          printf("Thread %d performed iteration %d\n",
31                   id, i);
32      }
33
34      printf("\n");
35      return 0;
36  }
```

Once you run this code, verify that the default behavior for this pragma is this sort of decomposition of iterations of the loop to threads, when you set the number of threads to 4 on the command line:



What happens when the number of iterations (16 in this code) is not evenly divisible by the number of threads? Try several cases to be certain how the compiler splits up the work. This type of decomposition is commonly used when accessing data that is stored in consecutive memory locations (such as an array) that might be cached by each thread.

7. Shared Data Decomposition Algorithm Strategy: one iteration per thread in a parallel for loop implementation strategy

file: openMP/07.parallelLoop-chunksOf1/parallelLoopChunksOf1.c

Build inside 07.parallelLoop-chunksOf1 directory:

```
make parallelLoopChunksOf1
```

Execute on the command line inside 07.parallelLoop-chunksOf1 directory:

```
./parallelLoopChunksOf1 4
Replace 4 with other values for the number of threads, or leave off
```

You can imagine other ways of assigning threads to iterations of a loop besides that shown above for four threads and 16 iterations. A simple decomposition sometimes used when your loop is not accessing consecutive memory locations would be to let each thread do one iteration, up to N threads, then start again with thread 0 taking the next iteration. This is declared in OpenMP using the pragma on line 31 of the following code. Also note that the commented code below it is an alternative explicit way of doing it. The schedule clause is the preferred style when using OpenMP and is more versatile, because you can easily change the *chunk size* that each thread will work on.

```
1  /* parallelLoopChunksOf1.c
2   * ... illustrates how to make OpenMP map threads to
3   *        parallel loop iterations in chunks of size 1
4   *        (use when not accesssing memory).
5   *
6   * Joel Adams, Calvin College, November 2009.
7   *
8   * Usage: ./parallelLoopChunksOf1 [numThreads]
9   *
10  * Exercise:
11  * 1. Compile and run, comparing output to source code,
12  *    and to the output of the 'equal chunks' version.
13  * 2. Uncomment the "commented out" code below,
14  *    and verify that both loops produce the same output.
15  *    The first loop is simpler but more restrictive;
16  *    the second loop is more complex but less restrictive.
17  */
18
19 #include <stdio.h>
20 #include <omp.h>
21 #include <stdlib.h>
22
23 int main(int argc, char** argv) {
24     const int REPS = 16;
25
26     printf("\n");
27     if (argc > 1) {
28         omp_set_num_threads( atoi(argv[1]) );
29     }
30
31     #pragma omp parallel for schedule(static,1)
32     for (int i = 0; i < REPS; i++) {
33         int id = omp_get_thread_num();
34         printf("Thread %d performed iteration %d\n",
35                 id, i);
36     }
37
38 /*
```

```
39      printf("\n---\n\n");
40
41      #pragma omp parallel
42      {
43          int id = omp_get_thread_num();
44          int numThreads = omp_get_num_threads();
45          for (int i = id; i < REPS; i += numThreads) {
46              printf("Thread %d performed iteration %d\n",
47                      id, i);
48          }
49      }
50  */
51      printf("\n");
52      return 0;
53  }
```

This can be made even more efficient if the next available thread simply takes the next iteration. In OpenMP, this is done by using *dynamic* scheduling instead of the static scheduling shown in the above code. Also note that the number of iterations, or chunk size, could be greater than 1 inside the schedule clause.

8. Coordination Using Collective Communication: Reduction

file: openMP/08.reduction/reduction.c

Build inside 08.reduction directory:

```
make reduction
```

Execute on the command line inside 08.reduction directory:

```
./reduction 4
Replace 4 with other values for the number of threads, or leave off
```

Once threads have performed independent concurrent computations, possibly on some portion of decomposed data, it is quite common to then *reduce* those individual computations into one value. This type of operation is called a **collective communication** pattern because the threads must somehow work together to create the final desired single value.

In this example, an array of randomly assigned integers represents a set of shared data (a more realistic program would perform a computation that creates meaningful data values; this is just an example). Note the common sequential code pattern found in the function called *sequentialSum* in the code below (starting line 51): a for loop is used to sum up all the values in the array.

Next let's consider how this can be done in parallel with threads. Somehow the threads must implicitly *communicate* to keep the overall sum updated as each of them works on a portion of the array. In the *parallelSum* function, line 64 shows a special clause that can be used with the parallel for pragma in OpenMP for this. 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 66.

```
1  /* reduction.c
2   * ... illustrates the OpenMP parallel-for loop's reduction clause
3   *
4   * Joel Adams, Calvin College, November 2009.
5   *
6   * Usage: ./reduction
7   *
8   * Exercise:
9   * - Compile and run.  Note that correct output is produced.
```

```
 * - Uncomment #pragma in function parallelSum(),
 *    but leave its reduction clause commented out
 * - Recompile and rerun.  Note that correct output is NOT produced.
 * - Uncomment 'reduction(+:sum)' clause of #pragma in parallelSum()
 * - Recompile and rerun.  Note that correct output is produced again.
 */

#include <stdio.h>   // printf()
#include <omp.h>     // OpenMP
#include <stdlib.h>  // rand()

void initialize(int* a, int n);
int sequentialSum(int* a, int n);
int parallelSum(int* a, int n);

#define SIZE 1000000

int main(int argc, char** argv) {
   int array[SIZE];

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

   initialize(array, SIZE);
   printf("\nSeq. sum: \t%d\nPar. sum: \t%d\n\n",
            sequentialSum(array, SIZE),
            parallelSum(array, SIZE) );

   return 0;
}

/* fill array with random values */
void initialize(int* a, int n) {
   int i;
   for (i = 0; i < n; i++) {
      a[i] = rand() % 1000;
   }
}

/* sum the array sequentially */
int sequentialSum(int* a, int n) {
   int sum = 0;
   int i;
   for (i = 0; i < n; i++) {
      sum += a[i];
   }
   return sum;
}

/* sum the array using multiple threads */
int parallelSum(int* a, int n) {
   int sum = 0;
   int i;
//   #pragma omp parallel for // reduction(+:sum)
   for (i = 0; i < n; i++) {
      sum += a[i];
   }
```

```
68     return sum;
69 }
```

Something to think about

Do you have an ideas about why the parallel for pragma without the reduction clause did not produce the correct result? Later examples will hopefully shed some light on this.

9. Coordination Using Collective Communication: Reduction revisited

Build inside 09.reduction-userDefined directory:

```
make reduction2
```

Execute on the command line inside 09.reduction-userDefined directory:

```
./reduction  4 4096
Replace 4 with other values for the number of threads
Replace 4096 with other values for n (computing up to n factorial)
```

The next example uses many threads to generate computations of factorials of n. Though there are likely other better ways to compute factorials, this example uses a very simple approach to illustrate how reduction can be used with the multiplication operation instead of addition in the previous example. The pragma for this is on line 34 in the code below, which also makes use of an additional C++ file, BigInt.h:

```
1  /* reduction2.cpp computes a table of factorial values,
2   *  using Owen Astrachan's BigInt class to explore
3   *  OpenMP's user-defined reductions.
4   *
5   *  Joel Adams, Calvin College, December 2015.
6   *
7   *  Usage: ./reduction2 [numThreads] [n]
8   *
9   *  Exercise:
10  *  - Build and run, record sequential time in a spreadsheet
11  *  - Uncomment #pragma omp parallel for directive, rebuild,
12  *     and read the error message carefully.
13  *  - Uncomment the #pragma omp declare directive, rebuild,
14  *     and note the user-defined * reduction for a BigInt.
15  *  - Rerun, using 2, 4, 6, 8, ... threads, recording
16  *     the times in the spreadsheet.
17  *  - Create a chart that plots the times vs the # of threads.
18  *  - Experiment with different n values
19  */
20
21 #include "BigInt.h"
22 #include <cassert>
23 #include <omp.h>
24
25 /*
26 #pragma omp declare reduction(*: BigInt: \
27                                omp_out = omp_out * omp_in) \
28                                initializer( omp_priv=BigInt(1))
29 */
30
31 BigInt factorial(unsigned n) {
```

```
32          BigInt result = 1;
33
34  //        #pragma omp parallel for reduction(*:result)
35          for (unsigned i = 2; i <= n; i += 1) {
36                  result *= i;
37          }
38
39          return result;
40  }
41
42  int main(int argc, char** argv) {  // on a 2GHz i7 CPU:
43          unsigned n = 4096;         // ~10 secs sequentially
44          unsigned numThreads = 1;
45
46          switch (argc) {
47                  case 3: n = atoi(argv[2]);
48                  case 2: numThreads = atoi(argv[1]);
49                  case 1: break;
50                  default: cout << "\nUsage: ./reduction2 [numThreads] [n]\n\n";
51          }
52          omp_set_num_threads(numThreads);
53
54          double startTime = omp_get_wtime();
55          BigInt nFactorial = factorial(n);
56          double time = omp_get_wtime() - startTime;
57
58          cout << "Computing " << n << "! using "
59               << numThreads  << " threads took: "
60               << time << " secs" << endl;
61
62          // run a few tests to validate the results
63          assert( factorial(0) == 1 );
64          assert( factorial(1) == 1 );
65          assert( factorial(2) == 2 );
66          assert( factorial(3) == 6 );
67          assert( factorial(4) == 24 );
68          assert( factorial(5) == 120 );
69          assert( factorial(32) == BigInt("263130836933693530167218012160000000") );
70          assert( factorial(100) == BigInt( string("933262154439441526816992388 5")
71                                                  + "626670049071596826438162146 8"
72                                                  + "592963895217599993229915608 9"
73                                                  + "414639761565182862536979208 2"
74                                                  + "722375825118521091686400000 0"
75                                                  + "000000000000000000" ) );
76          cout << "All tests passed!\n" << flush;
77  }
```

With this code you can begin to explore the time it takes to execute the program when using increasing numbers of threads for various values of n. Follow the instructions at the top of the file.

10. Dynamic Data Decomposition

Build inside 10.parallelLoop-dynamicSchedule directory:

```
make dynamicScheduling
```

Execute on the command line inside 10.parallelLoop-dynamicSchedule directory:

```
./dynamicScheduling 4
Replace 4 with other values for the number of threads
```

The following example computes factorials for the numbers 2 through 512, placing the result in an array. This array of results is the data in this data decomposition pattern. Since each number will take a different amount of time to compute, this is a case where using dynamic scheduling of the work improves the performance. Try the tasks lsited in the header of the code shown below to see this.

```
/* dynamicScheduling.cpp computes a table of factorial values,
 *  using Owen Astrachan's BigInt class to explore
 *  OpenMP's schedule() clause.
 *
 *  @author: Joel Adams, Calvin College, Dec 2015.
 *
 *  Usage: ./dynamicScheduling [numThreads]
 *
 *  Exercise:
 *  - Build and run, record sequential run time in a spreadsheet
 *  - Uncomment #pragma omp parallel for, rebuild,
 *      run using 2, 4, 6, 8, ... threads, record run times.
 *  - Uncomment schedule(dynamic), rebuild,
 *      run using 2, 4, 6, 8, ... threads, record run times.
 *  - Create a line chart plotting run times vs # of threads.
 */

#include "BigInt.h"                 // class BigInt
#include <omp.h>                    // OpenMP functions
#include <cassert>                  // assert()

/* factorial(n) computes n!
 * @param: n, an unsigned int.
 * @return: n!, a BigInt.
 */
BigInt factorial(unsigned n) {
        BigInt result = 1;              // 0! or 1!

        for (unsigned i = 2; i <= n; ++i) {
                result *= i;
        }

        return result;
}

int main(int argc, char** argv) {           // on a 2 GHz i7 CPU:
        const unsigned MAX = 512;           //  ~14 secs sequentially
//        const unsigned MAX = 800;           //  ~60 secs sequentially
        BigInt factorialTable[MAX+1];

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

        cout << "\nDepending on the speed of your computer,"
             << "\n  this program may take a while to complete,"
             << "\n  so please wait patiently...\n" << endl;

        double startTime = omp_get_wtime();
//        #pragma omp parallel for // schedule(dynamic)
        for (unsigned i = 0; i <= MAX; i++) {
                factorialTable[i] = factorial(i);
```

```
51          }
52          double totalTime = omp_get_wtime() - startTime;
53
54          cout << "Computing 0! .. " << MAX << "! took: "
55               << totalTime << " secs\n" << endl;
56
57          // run a few tests to validate the results
58          assert( factorialTable[0] == 1 );
59          assert( factorialTable[1] == 1 );
60          assert( factorialTable[2] == 2 );
61          assert( factorialTable[3] == 6 );
62          assert( factorialTable[4] == 24 );
63          assert( factorialTable[5] == 120 );
64          assert( factorialTable[32] == BigInt("263130836933693530167218012160000000") );
65          assert( factorialTable[100] == BigInt( string("933262154439441526816992388 5")
66                                                 + "626670049071596826438162146 8"
67                                                 + "592963895217599993229915608 9"
68                                                 + "414639761565182862536979208 2"
69                                                 + "722375825118521091686400000 0"
70                                                 + "000000000000000000" ) );
71          cout << "All tests passed!\n" << endl;
72  }
```

3.2.3 Patterns used when threads share data values

11. Shared Data Algorithm Strategy: Parallel-for-loop pattern needs non-shared, private variables

file: openMP/11.private/private.c

Build inside 11.private directory:

```
make private
```

Execute on the command line inside 11.private directory:

```
./private
```

In this example, you will try a parallel for loop where additional variables (i, j in the code) cannot be shared by all of the threads, but must instead be *private* to each thread, which means that each thread has its own copy of that variable. In this case, the outer loop is being split into chunks and given to each thread, but the inner loop is being executed by each thread for each of the elements in its chunk. The loop counting variables must be maintained separately by each thread. Because they were initially declared outside the loops at the begininning of the program, by default these variables are shared by all the threads.

```
1   /* private.c
2    * ... illustrates why private variables are needed with OpenMP's parallel for loop
3    *
4    * Joel Adams, Calvin College, November 2009.
5    *
6    * Usage: ./private
7    *
8    * Exercise:
9    * - Run, noting that the sequential program produces correct results
10   * - Uncomment line A, recompile/run and compare
11   * - Recomment line A, uncomment line B, recompile/run and compare
12   */
13
```

```c
#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];

    printf("\n");
    // set all array entries to 1
//    #pragma omp parallel for                    // A
//    #pragma omp parallel for private(i,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;
}
```

12. Race Condition: missing the mutual exclusion coordination pattern

file: openMP/12.mutualExclusion-atomic/atomic.c

Build inside 12.mutualExclusion-atomic directory:

```
make atomic
```

Execute on the command line inside 12.mutualExclusion-atomic directory:

```
./atomic
```

When a variable must be shared by all the threads, as in this example below, an issue called a *race condition* can occur when the threads are updating that variable concurrently. This happens because there are multiple underlying machine instructions needed to complete the update of the memory location and each thread must execute all of them atomically before another thread does so, thus ensuring **mutual exclusion** between the threads when updating a shared variable. This is done using the OpenMP pragma shown in this code.

```c
/* atomic.c
 * ... illustrates a race condition when multiple threads read from /
 *  write to a shared variable (and explores OpenMP atomic operations).
```

```
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./atomic
 *
 * Exercise:
 *  - Compile and run 10 times; note that it always produces the correct balance: $1,000,000.00
 *  - To parallelize, uncomment A, recompile and rerun multiple times, compare results
 *  - To fix: uncomment B, recompile and rerun, compare
 */

#include <stdio.h>  // printf()
#include <omp.h>    // OpenMP

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                    // A
    for (i = 0; i < REPS; i++) {
        // #pragma omp atomic                      // B
        balance += 1.0;
    }

    printf("\nAfter %d $1 deposits, your balance is $%0.2f\n\n",
                REPS, balance);

    return 0;
}
```

13. The Mutual Exclusion Coordination Pattern: two ways to ensure

file: openMP/13.mutualExclusion-critical/critical.c

Build inside 13.mutualExclusion-critical directory:

```
make critical
```

Execute on the command line inside 13.mutualExclusion-critical directory:

```
./critical
```

Here is another way to ensure **mutual exclusion** in OpenMP.

```
/* critical.c
 * ... fixes a race condition when multiple threads read from /
 *  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 correct balance $1,000,000.00
 *  - Comment out A; recompile/run, and note incorrect result
 *  - To fix: uncomment B1+B2+B3, 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                          // A
//        #pragma omp critical                      // B1
//        {                                         // B2
        balance += 1.0;
//        }                                         // B3
    }

    printf("\nAfter %d $1 deposits, your balance is %0.2f\n",
                REPS, balance);

    return 0;
}
```

14. Mutual Exclusion Coordination Pattern: compare performance

file: openMP/14.mutualExclusion-critical2/critical2.c

Build inside 14.mutualExclusion-critical2 directory:

```
make critical2
```

Execute on the command line inside 14.mutualExclusion-critical2 directory:

```
./critical2
```

Here is an example of how to compare the performance of using the atomic pragma directive and the critical pragma directive. Note that there is a function in OpenMP that lets you obtain the current time, which enables us to determine how long it took to run a particular section of our program.

```
/* critical2.c
 * ... compares the performance of OpenMP's critical and atomic directives
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./critical2
 *
 * Exercise:
 *  - Compile, run, compare times for critical vs. atomic
 *  - Note how much more costly critical is than atomic
 *  - Research: Create an expression that, when assigned to balance,
```

```
 *     critical can handle but atomic cannot
 */

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

void print(char* label, int reps, double balance, double total, double average) {
    printf("\nAfter %d $1 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,
           atomicTime = 0.0,
           criticalTime = 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 += 1.0;
    }
    stopTime = omp_get_wtime();
    atomicTime = stopTime - startTime;
    print("atomic", REPS, balance, atomicTime, atomicTime/REPS);


    // simulate the same number of deposits using critical
    balance = 0.0;
    startTime = omp_get_wtime();
    #pragma omp parallel for
    for (i = 0; i < REPS; i++) {
         #pragma omp critical
         {
             balance += 1.0;
         }
    }
    stopTime = omp_get_wtime();
    criticalTime = stopTime - startTime;
    print("critical", REPS, balance, criticalTime, criticalTime/REPS);

    printf("criticalTime / atomicTime ratio: %0.12f\n\n",
             criticalTime / atomicTime);

    return 0;
}
```

15. Mutual Exclusion Coordination Pattern: language difference

file: openMP/15.mutualExclusion-critical3/critical3.c

Build inside 15.mutualExclusion-critical3 directory:

```
make critical3
```

Execute on the command line inside 15.mutualExclusion-critical3 directory:

```
./critical3
```

The following is a C++ code example to illustrate some language differences between C and C++. Try the exercises described in the code below.

```
/* critical3.c
 * ... a simple case where OpenMP's critical works but atomic does not.
 *
 * Joel Adams, Calvin College, November 2009.
 *
 * Usage: ./critical3
 *
 * Exercise:
 *  - Compile, run, note resulting output is correct.
 *  - Uncomment line A, recompile, rerun, note results.
 *  - Uncomment line B, recompile, note results.
 *  - Recomment line B, uncomment line C, recompile,
 *    rerun, note change in results.
 */

#include<iostream>   // cout
#include<omp.h>      // openmp
using namespace std;

int main(int argc, char** argv) {
    cout << "\n";

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

//    #pragma omp parallel                          // A
    {
        int id = omp_get_thread_num();
        int numThreads = omp_get_num_threads();
//        #pragma omp atomic                        // B
//        #pragma omp critical                      // C
        cout << "Hello from thread #" << id
             << " out of " << numThreads
             << " threads.\n";
    }

    cout << "\n";
}
```

Some Explanation

A C line like this:

printf("Hello from thread #%d of %dn", id, numThreads);

is a single function call that is pretty much performed atomically, so you get pretty good output like.

```
Hello from thread #0 of 4
Hello from thread #2 of 4
Hello from thread #3 of 4
Hello from thread #1 of 4
```

By contrast, the C++ line:

cout << "Hello from thread #" << id << " of " << numThreads << endl;

has 5 different function calls, so the outputs from these functions get interleaved within the shared stream cout as the threads 'race' to write to it. You may have observed output similar to this:

Hello from thread #Hello from thread#Hello from thread#0 of 4Hello from thread#

2 of 43 of 4

1 of 4

The other facet that this particular patternlet shows is that OpenMP's atomic directive will not fix this – it is too complex for atomic, so the compiler flags that as an error. To make this statement execute indivisibly, you need to use the critical directive, providing a pretty simple case where critical works and atomic does not.

3.2.4 Task Decomposition Algorithm Strategies

All threaded programs have some form of task decomposition, that is, delineating which threads will do what tasks in parallel at certain points in the program. We have seen one way of dictating this by using the master-worker implementation, where one thread does one task and all the others to another. Here we introduce a more general approach that can be used.

16. Task Decomposition Algorithm Strategy using OpenMP section directive

file: openMP/16.sections/sections.c

Build inside 16.sections directory:

```
make sections
```

Execute on the command line inside 16.sections directory:

```
./sections
```

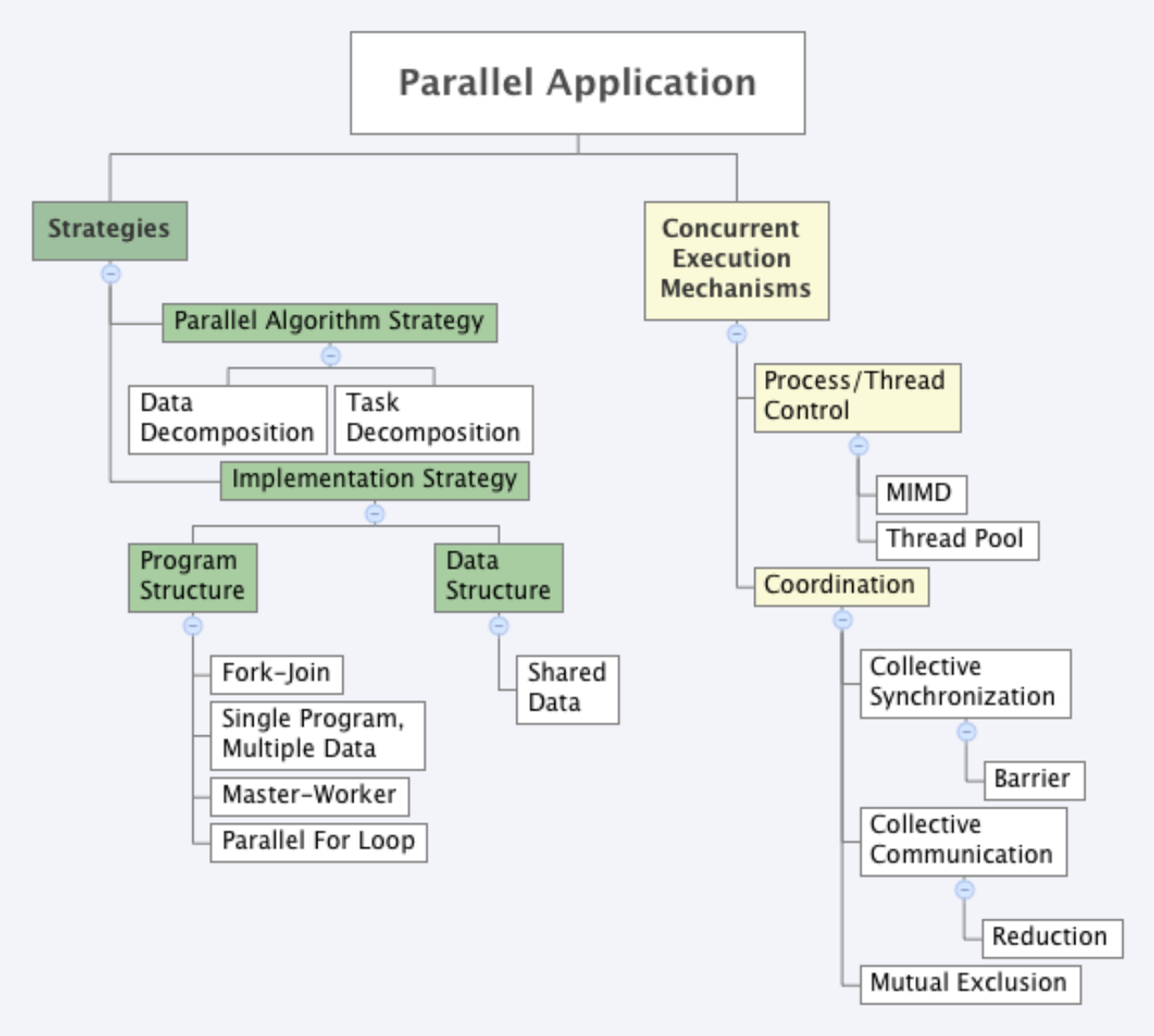
This example shows how to create a program with arbitrary separate tasks that run concurrently. This is useful if you have tasks that are not dependent on one another.

```
1  /* sections.c
2   * ... illustrates the use of OpenMP's parallel section/sections directives,
3   *      which can be used for task parallelism...
4   *
5   * Joel Adams, Calvin College, November 2009.
6   *
7   * Usage: ./sections
8   *
9   * Exercise: Compile, run (several times), compare output to source code.
10  */
11
12 #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("Task/section A performed by thread %d\n",
                    omp_get_thread_num() );
        }
        #pragma omp section
        {
            printf("Task/section B performed by thread %d\n",
                    omp_get_thread_num() );
        }
        #pragma omp section
        {
            printf("Task/section C performed by thread %d\n",
                    omp_get_thread_num() );
        }
        #pragma omp section
        {
                printf("Task/section D performed by thread %d\n",
                        omp_get_thread_num() );
        }
    }

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

    return 0;
}
```

3.2.5 Categorizing Patterns

There has been a fair amount of work by several researchers who have catergorized patterns found in parallel programs. We have shown you simple examples of several of them that are very common when writing OpenMP programs that use shared memory. Now that you have seen them, you can try to imagine the patterns falling into the categories shown on the following diagram:



Most programs you will write will include patterns for an Algorithm Strategy (both data decomposition and task decomposition), some of the Implementation Strategies (if not all), and some of the Coordination Mechanisms. The patternlets show simple examples that you can use as a guide. In OpenMP, most programs use various shared data in memory as their data structure implementation strategy. The Process/Thread Control Mechanism patterns are built in to any OpenMP program. Multiple Instruction, Multiple Data (MIMD) is built in because forked threads operate independently on different data. Likewise, pools of threads are part of every compiled OpenMP threaded program.