**CSinParallel Project**

June 08, 2016

1. [Parallel Programming Patterns](#_bookmark0) 2
   1. [An organization of parallel patterns](#_bookmark1) 2
2. [Message Passing Parallel Patternlets](#_bookmark2) 4
   1. [Source Code](#_bookmark3) 4
   2. [00. Single Program, Multiple Data](#_bookmark4) 4
   3. [01. The Master-Worker Implementation Strategy Pattern](#_bookmark5) 5
   4. [02. Message passing 1, using Send-Receive of a single value](#_bookmark6) 6
   5. [03. Message passing 2, using Send-Receive of an array of values](#_bookmark7) 7
   6. [04. Message passing 3, using Send-Receive with master-worker pattern](#_bookmark8) 8
   7. [05. Data Decomposition: on equal-sized chunks using parallel-for](#_bookmark9) 9
   8. [06. Data Decomposition: on chunks of size 1 using parallel-for](#_bookmark10) 10
   9. [07. Broadcast: a special form of message passing](#_bookmark11) 11
   10. [08. Broadcast: send data to all processes](#_bookmark12) 12
   11. [09. Collective Communication: Reduction](#_bookmark13) 13
   12. [10. Collective Communication: Reduction](#_bookmark14) 14
   13. [11. Collective communication: Scatter for message-passing data decomposition](#_bookmark15) 16
   14. [12. Collective communication: Gather for message-passing data decomposition](#_bookmark16) 17
   15. [13. The Barrier Coordination Pattern](#_bookmark17) 18
   16. [14. Timing code using the Barrier Coordination Pattern](#_bookmark18) 20
   17. [15. Sequence Numbers](#_bookmark19) 21
3. [Shared Memory Parallel Patternlets in OpenMP](#_bookmark20) 24
   1. [Source Code](#_bookmark21) 24
   2. [Patternlets Grouped By Type](#_bookmark22) 25

**Parallel Patternlets,**

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).

# 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. **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 implemen- tation 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. **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.

**Parallel Patternlets,**

* + - 1. *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 dis- tributed 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:
3. MPI, or Message Passing Interface, for message passing.
4. 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 mem- ory 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*](#_bookmark2)

[*Shared Memory Parallel Patternlets in OpenMP*](#_bookmark20)

**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.

# 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.

# 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*

* 1. **01. 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/01.masterWorker/masterWorker.c*

* 1. **02. 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("**\n**Please run this program using -np N where N is positive and even.**\n\n**");

}

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/02.messagePassing/messagePassing.c*

* 1. **03. 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("**\n**Process %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("**\n**Please run this program using -np N where N is positive and even.**\n\n**");

}

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/03.messagePassing2/messagePassing2.c*

* 1. **04. 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("**\n**Please run this program with at least 2 processes**\n\n**");

}

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/04.messagePassing3/messagePassing3.c*

* 1. **05. 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/05.parallelLoop-equalChunks/parallelLoopEqualChunks.c*

* 1. **06. 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/06.parallelLoop-chunksOf1/parallelLoopChunksOf1.c*

* 1. **07. Broadcast: a special form of message passing**

This example shows how a data item read from a file can be sent to all the processes.

*/\* broadcast.c*

* + - *... illustrates the use of MPI\_Bcast() with a scalar value...*
    - *(compare to array version).*
    - *Joel Adams, Calvin College, April 2016.*

*\**

* + - *Usage: mpirun -np N ./broadcast*

*\**

* + - *Exercise:*
    - *- Compile and run several times,*
    - *using 2, 4, and 8 processes*
    - *- Use source code to trace execution and output*
    - *(noting contents of file "data.txt");*
    - *- Explain behavior/effect of MPI\_Bcast().*

*\*/*

*#include <mpi.h>*

*#include <stdio.h>*

*#include <stdlib.h>*

*#include <assert.h>*

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

**int** answer = 0;

**int** numProcs = 0, myRank = 0;

MPI\_Init(&argc, &argv); MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcs); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &myRank);

**if** (myRank == 0) {

**FILE** \*filePtr = fopen("data.txt", "r"); assert( filePtr != NULL ); fscanf(filePtr, " %d", &answer); fclose(filePtr);

}

printf("BEFORE the broadcast, process %d’s answer = %d**\n**", myRank, answer);

MPI\_Bcast(&answer, 1, MPI\_INT, 0, MPI\_COMM\_WORLD); printf("AFTER the broadcast, process %d’s answer = %d**\n**",

myRank, answer);

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/07.broadcast/broadcast.c*

* 1. **08. Broadcast: send data to all processes**

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

*/\* broadcast2.c*

* + - *... illustrates the use of MPI\_Bcast() for arrays...*
    - *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 some arbitrary values*

* *@param: a, an int\*.*
* *@param: size, an int.*
* *Precondition: a is the address of an array of ints.*
* *&& size is the number of ints a can hold.*
* *Postcondition: a has been filled with arbitrary values*

*\* { 11, 12, 13, ... }.*

*\*/*

**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*

* *@param: str, a char\**
* *@param: id, an int*
* *@param: a, an int\*.*
* *Precondition: str points to either "BEFORE" or "AFTER"*
* *&& id is the rank of this MPI process*
* *&& a is the address of an 8-element int array.*
* *Postcondition: str, id, and a have all been written to stdout.*

*\*/*

**void** print(**char**\* str, **int** id, **int**\* a) {

printf("%s broadcast, process %d has: {%d, %d, %d, %d, %d, %d, %d, %d}**\n**", str, id, 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("BEFORE", myRank, array);

MPI\_Bcast(array, MAX, MPI\_INT, 0, MPI\_COMM\_WORLD);

print("AFTER", myRank, array); MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/08.broadcast2/broadcast2.c*

* 1. **09. 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 calcu- lation 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("**\n**The 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*

* 1. **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("**\n**Before 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("**\n**After 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*

* 1. **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*

* 1. **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*

* 1. **13. The Barrier Coordination Pattern**

A barrier is used when you want all the processes to complete a portion of code before continuing. Use this exercise to verify that is is ocurring when you add the call to the MPI\_Barrier funtion.

*/\* barrier.c*

* + - *... illustrates the behavior of MPI\_Barrier() ...*

*\**

* + - *Joel Adams, Calvin College, May 2013.*
    - *Bill Siever, April 2016*
    - *(Converted to master/worker pattern).*
    - *Joel Adams, April 2016*
    - *(Refactored code so that just one barrier needed).*

*\**

* + - *Usage: mpirun -np 8 ./barrier*

*\**

* + - *Exercise:*
    - *- Compile; then run the program several times,*
    - *noting the interleaved 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*

*/\* Have workers send messages to the master, which prints them.*

* *@param: id, an int*
* *@param: numProcesses, an int*
* *@param: hostName, a char\**
* *@param: position, a char\**

*\**

* *Precondition: this function is being called by an MPI process*
* *&& id is the MPI rank of that process*
* *&& numProcesses is the number of processes in the computation*
* *&& hostName points to a char array containing the name of the*
* *host on which this MPI process is running*
* *&& position points to "BEFORE" or "AFTER".*

*\**

* *Postcondition: each process whose id > 0 has sent a message to process 0*
* *containing id, numProcesses, hostName, and position*
* *&& process 0 has received and output each message.*

*\*/*

*#define BUFFER\_SIZE 200*

*#define MASTER* *0*

**void** sendReceivePrint(**int** id, **int** numProcesses, **char**\* hostName, **char**\* position) {

**char** buffer[BUFFER\_SIZE] = {’\0’};; MPI\_Status status;

**if** (id != MASTER) {

*// Worker: Build a message and send it to the Master*

**int** length = sprintf(buffer,

"Process #%d of %d on %s is %s the barrier.**\n**", id, numProcesses, hostName, position);

MPI\_Send(buffer, length+1, MPI\_CHAR, 0, 0, MPI\_COMM\_WORLD);

} **else** {

*// Master: Receive and print the messages from all Workers*

**for**(**int** i = 0; i < numProcesses-1; i++) {

MPI\_Recv(buffer, BUFFER\_SIZE, MPI\_CHAR, MPI\_ANY\_SOURCE,

MPI\_ANY\_TAG, MPI\_COMM\_WORLD, &status); printf(buffer);

}

}

}

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

**int** id = -1, numProcesses = -1, length = -1;

**char** myHostName[MPI\_MAX\_PROCESSOR\_NAME] = {’\0’};

MPI\_Init(&argc, &argv); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id); MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcesses); MPI\_Get\_processor\_name (myHostName, &length);

sendReceivePrint(id, numProcesses, myHostName, "BEFORE");

*// MPI\_Barrier(MPI\_COMM\_WORLD);*

sendReceivePrint(id, numProcesses, myHostName, "AFTER");

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/13.barrier/barrier.c*

* 1. **14. Timing code using the Barrier Coordination Pattern**

In this example you can run the code several times and determine the average, median, and minimum execution time when the code has a barrier and when it does not. The primary purpose of this exercise is to illustrate that one of the most useful uses of a barrier is to ensure that you are getting legitimate timings for your code examples. By using a barrier, you ensure that all processes have finished before recording the time using the master node.

*/\* barrier+timing.c*

* + - *... illustrates the behavior of MPI\_Barrier()*
    - *to coordinate process-timing.*

*\**

* + - *Joel Adams, April 2016*

*\**

* + - *Usage: mpirun -np 8 ./barrier+timing*

*\**

* + - *Exercise:*
    - *- Compile; then run the program five times,*
    - *- In a spreadsheet, compute the average,*
    - *median, and minimum of the five times.*
    - *- Uncomment the two MPI\_Barrier() calls;*
    - *then recompile, rerun five times, and*
    - *compute the new average, median, and min*
    - *times.*
    - *- Why did uncommenting the barrier calls*
    - *produce the change you observed?*

*\*/*

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

*#include <mpi.h> // MPI*

*#include <unistd.h> // sleep()*

*#define MASTER 0*

*/\* answer the ultimate question of life, the universe,*

* *and everything, based on id and numProcs.*
* *@param: id, an int*
* *@param: numProcs, an int*
* *Precondition: id is the MPI rank of this process*
* *&& numProcs is the number of MPI processes.*
* *Postcondition: The return value is 42.*

*\*/*

**int** solveProblem(**int** id, **int** numProcs) { sleep( ((**double**)id+1) / numProcs);

**return** 42;

}

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

**int** id = -1, numProcesses = -1;

**double** startTime = 0.0, totalTime = 0.0;

**int** answer = 0.0; MPI\_Init(&argc, &argv);

MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id); MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcesses);

*// MPI\_Barrier(MPI\_COMM\_WORLD);*

**if** ( id == MASTER) {

startTime = MPI\_Wtime();

}

answer = solveProblem(id, numProcesses);

*// MPI\_Barrier(MPI\_COMM\_WORLD);*

**if** ( id == MASTER ) {

totalTime = MPI\_Wtime() - startTime;

printf("**\n**The answer is %d; computing it took %f secs.**\n\n**", answer, totalTime);

}

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/14.barrier+Timing/barrier+timing.c*

* 1. **15. Sequence Numbers**

Tags can be placed on messages that are sent from a non-master node and received by the master node. Using tags is an alternative form of simulating the barrier example in example 13 above.

*/\* sequenceNumbers.c*

* + - *... shows how to acheive barrier-like behavior*
    - *by using MPI message tags as sequence numbers.*

*\**

* + - *Joel Adams, Calvin College, April 2016.*

*\**

* + - *Usage: mpirun -np 8 ./sequenceNumbers*

*\**

* + - *Exercise:*
    - *1. Compile; then run the program several times,*
    - *noting the intermixed outputs*
    - *2. Comment out the sendReceivePrint(..., "SECOND", 1); call;*
    - *uncomment the sendReceivePrint(..., "SECOND", 2); call;*
    - *then recompile and rerun, noting how the output changes.*
    - *3. Uncomment the sendReceivePrint(..., "THIRD", 3);*
    - *and sendReceivePrint(..., "FOURTH", 4); calls,*
    - *then recompile and rerun, noting how the output changes.*
    - *4. Explain the differences: what has caused the changes*
    - *in the program’s behavior, and why?*

*\*/*

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

*#include <mpi.h> // MPI*

*/\* Have workers send messages to the master, which prints them.*

* *@param: id, an int*
* *@param: numProcesses, an int*
* *@param: hostName, a char\**
* *@param: messageNum, a char\**
* *@param: tagValue, an int*

*\**

* *Precondition: this routine is being called by an MPI process*
* *&& id is the MPI rank of that process*
* *&& numProcesses is the number of processes in the computation*
* *&& hostName points to a char array containing the name of the*
* *host on which this MPI process is running*
* *&& messageNum is "FIRST", "SECOND", "THIRD", ...*
* *&& tagValue is the value for the tags of the message*
* *being sent and received this invocation of the function.*

*\**

* *Postcondition: each process whose id > 0 has sent a message to process 0*
* *containing id, numProcesses, hostName, messageNum,*
* *and tagValue*
* *&& process 0 has received and output each message.*

*\*/*

*#define BUFFER\_SIZE 200*

*#define MASTER* *0*

**void** sendReceivePrint(**int** id, **int** numProcesses, **char**\* hostName,

**char**\* messageNum, **int** tagValue) {

**char** buffer[BUFFER\_SIZE] = {’\0’};; MPI\_Status status;

**if** (id != MASTER) {

*// Worker: Build a message and send it to the Master*

**int** length = sprintf(buffer,

"This is the %s message from process #%d of %d on %s.**\n**", messageNum, id, numProcesses, hostName);

MPI\_Send(buffer, length+1, MPI\_CHAR, 0, tagValue, MPI\_COMM\_WORLD);

} **else** {

*// Master: Receive and print the messages from all Workers*

**for**(**int** i = 0; i < numProcesses-1; i++) {

MPI\_Recv(buffer, BUFFER\_SIZE, MPI\_CHAR, MPI\_ANY\_SOURCE,

tagValue, MPI\_COMM\_WORLD, &status); printf(buffer);

}

}

}

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

**int** id = -1, numProcesses = -1, length = -1;

**char** myHostName[MPI\_MAX\_PROCESSOR\_NAME] = {’\0’};

MPI\_Init(&argc, &argv); MPI\_Comm\_rank(MPI\_COMM\_WORLD, &id); MPI\_Comm\_size(MPI\_COMM\_WORLD, &numProcesses); MPI\_Get\_processor\_name (myHostName, &length);

sendReceivePrint(id, numProcesses, myHostName, "FIRST", 1); sendReceivePrint(id, numProcesses, myHostName, "SECOND", 1);

*// sendReceivePrint(id, numProcesses, myHostName, "SECOND", 2);*

*// sendReceivePrint(id, numProcesses, myHostName, "THIRD", 3);*

*// sendReceivePrint(id, numProcesses, myHostName, "FOURTH", 4);*

MPI\_Finalize();

**return** 0;

}

*file: patternlets/MPI/15.sequenceNumbers/sequenceNumbers.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.

# 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.

* 1. **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*](#_bookmark23)

[*Data Decomposition Algorithm Strategies and Related Coordination Strategies*](#_bookmark24)[*Patterns used when threads share data values*](#_bookmark25)

[*Task Decomposition Algorithm Strategies*](#_bookmark26)[*Categorizing Patterns*](#_bookmark27)

* + 1. **Shared Memory Program Structure and Coordination Patterns**

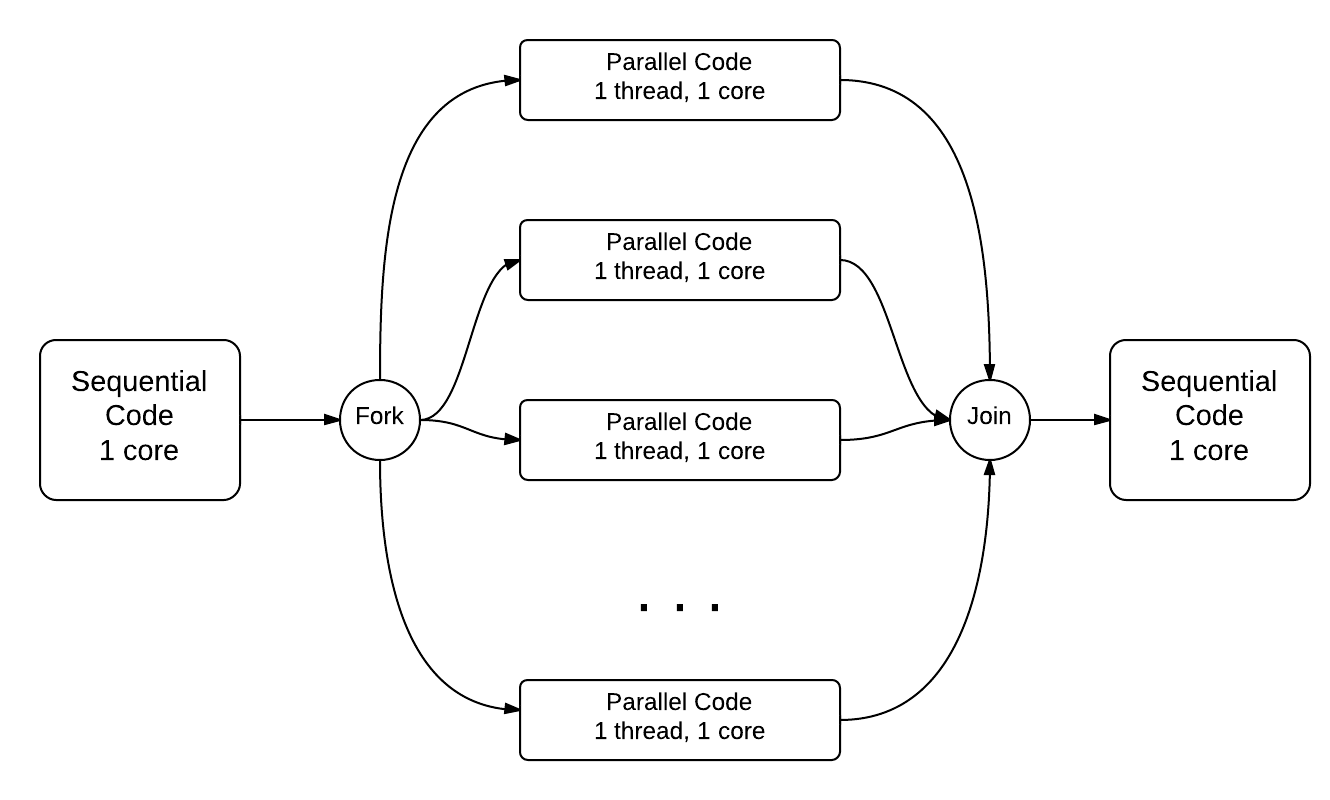
1. **Program Structure Implementation Strategy: The basic fork-join pattern**

*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.

1 */\* forkJoin.c*

2 *\* ... illustrates the fork-join pattern*

3 *\* using OpenMP’s parallel directive.*

4 *\**

5 *\* Joel Adams, Calvin College, November 2009.*

6 *\**

7 *\* Usage: ./forkJoin*

8 *\**

9 *\* Exercise:*

10 *\* - Compile & run, uncomment the pragma,*

11 *\* recompile & run, compare results.*

12 *\*/*

13

14 *#include <stdio.h> // printf()*

15 *#include <omp.h> // OpenMP*

16

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

18

19 printf("**\n**Before...**\n**");

20

21 *// #pragma omp parallel*

22 printf("**\n**During...");

23

24 printf("**\n\n**After...**\n\n**");

25

26 **return** 0;

27 }

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.

1 */\* forkJoin2.c*

2 *\* ... illustrates the fork-join pattern*

3 *\* using multiple OpenMP parallel directives,*

4 *\* and changing the number of threads two ways.*

5 *\**

6 *\* Joel Adams, Calvin College, May 2013.*

7 *\**

8 *\* Usage: ./forkJoin2*

9 *\**

10 *\* Exercise:*

11 *\* - Compile & run, compare results to source.*

12 *\* - Predict how many threads will be used in ’Part IV’?*

13 *\* - Uncomment ’Part IV’, recompile, rerun.*

14 *\*/*

15

16 *#include <stdio.h> // printf()*

17 *#include <omp.h> // OpenMP*

18

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

|  |  |  |
| --- | --- | --- |
| 20 |  | |
| 21 |  | printf("**\n**Beginning**\n**"); |
| 22 |  |  |
| 23 |  | *#pragma omp parallel* |
| 24 |  | printf("**\n**Part I"); |
| 25 |  |  |
| 26 |  | printf("**\n\n**Between I and II...**\n**"); |
| 27 |  |  |
| 28 |  | omp\_set\_num\_threads(3); |
| 29 |  |  |
| 30 |  | *#pragma omp parallel* |
| 31 |  | printf("**\n**Part II..."); |
| 32 |  |  |
| 33 |  | printf("**\n\n**Between II and III...**\n**"); |
| 34 |  |  |
| 35 |  | *#pragma omp parallel num\_threads(5)* |
| 36 |  | printf("**\n**Part III..."); |
| 37 | */\** |  |
| 38 |  | *printf("\n\nBetween III and IV...\n");* |
| 39 |  |  |
| 40 |  | *#pragma omp parallel* |
| 41 |  | *printf("\nPart IV...");* |
| 42 | *\*/* |  |
| 43 |  | printf("**\n\n**End**\n\n**"); |
| 44 |  |  |
| 45 |  | **return** 0; |
| 46 | } |  |

1. **Program Structure Implementation Strategy: Single Program, multiple data**

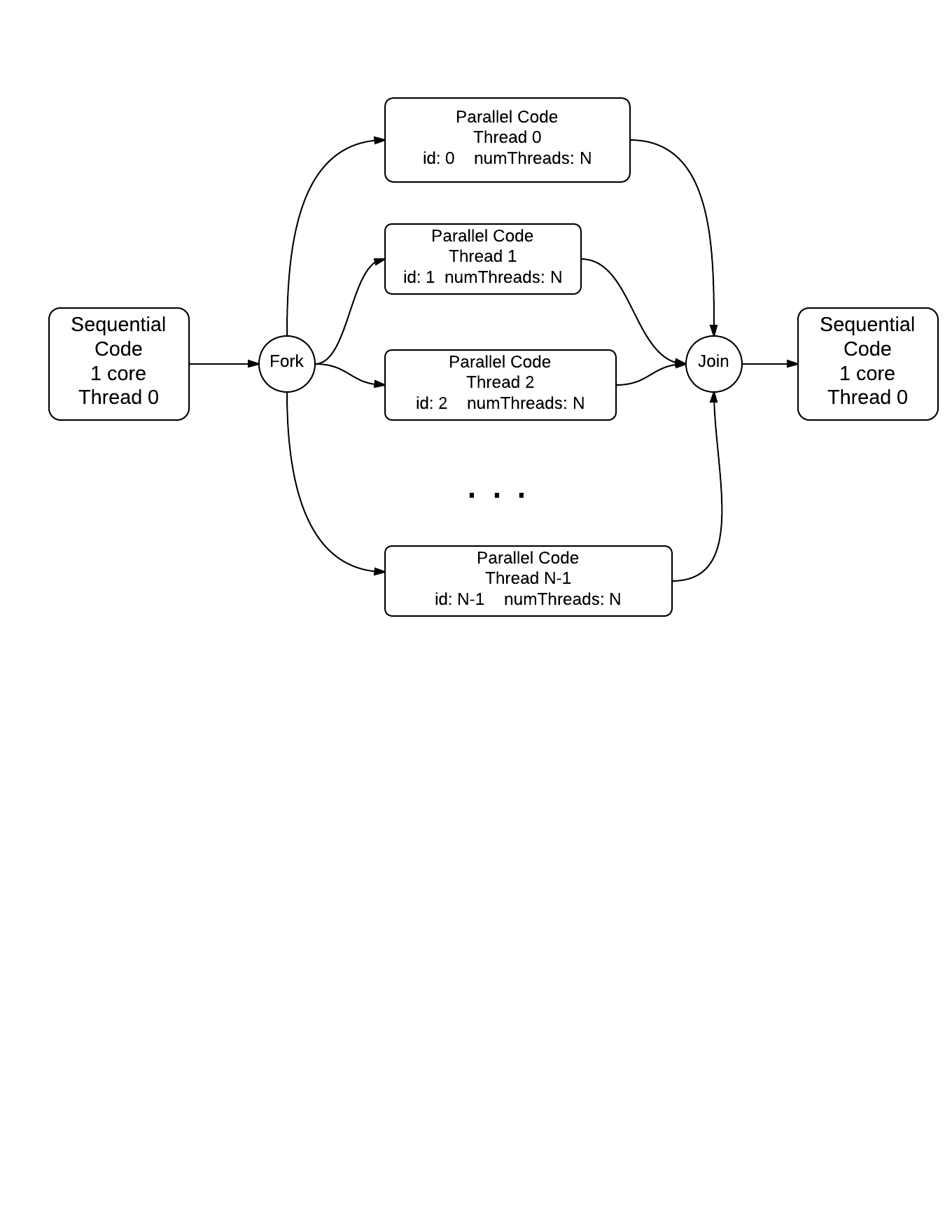
*file: openMP/02.spmd/spmd.c Build inside 02.spmd directory:* make spmd

*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.

1 */\* spmd.c*

2 *\* ... illustrates the single-program-multiple-data (SPMD)*

3 *\* pattern using two basic OpenMP commands...*

4 *\**

5 *\* Joel Adams, Calvin College, November 2009.*

6 *\**

7 *\* Usage: ./spmd*

8 *\**

9 *\* Exercise:*

10 *\* - Compile & run*

11 *\* - Uncomment pragma, recompile & run, compare results*

12 *\*/*

13

14 *#include <stdio.h>*

15 *#include <omp.h>*

16

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

18 printf("**\n**");

19

20 *// #pragma omp parallel*

21 {

22 **int** id = omp\_get\_thread\_num();

23 **int** numThreads = omp\_get\_num\_threads();

24 printf("Hello from thread %d of %d**\n**", id, numThreads);

25 }

26

27 printf("**\n**");

28 **return** 0;

29 }

1. **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.

1 */\* spmd2.c*

2 *\* ... illustrates the SPMD pattern in OpenMP,*

3 *\* using the commandline arguments*

4 *\* to control the number of threads.*

5 *\**

6 *\* Joel Adams, Calvin College, November 2009.*

7 *\**

8 *\* Usage: ./spmd2 [numThreads]*

9 *\**

10 *\* Exercise:*

11 *\* - Compile & run with no commandline args*

12 *\* - Rerun with different commandline args,*

13 *\* until you see a problem with thread ids*

14 *\* - Fix the race condition*

15 *\* (if necessary, compare to 02.spmd)*

16 *\*/*

17

18 *#include <stdio.h>*

19 *#include <omp.h>*

20 *#include <stdlib.h>*

21

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

23 **int** id, numThreads;

24

25 printf("**\n**");

26 **if** (argc > 1) {

27 omp\_set\_num\_threads( atoi(argv[1]) );

28 }

29

30 *#pragma omp parallel*

31 {

32 id = omp\_get\_thread\_num();

33 numThreads = omp\_get\_num\_threads();

ello

|  |  |  |
| --- | --- | --- |
| 34 |  | printf("H |
| 35 |  | } |
| 36 |  |  |
| 37 |  | printf("**\n**"); |
| 38 |  | **return** 0; |
| 39 | } |  |

from thread %d of %d**\n**", id, numThreads);

1. **Coordination: Synchronization with a Barrier**

*file: openMP/04.barrier/barrier.c Build inside 04.barrier directory:* make barrier

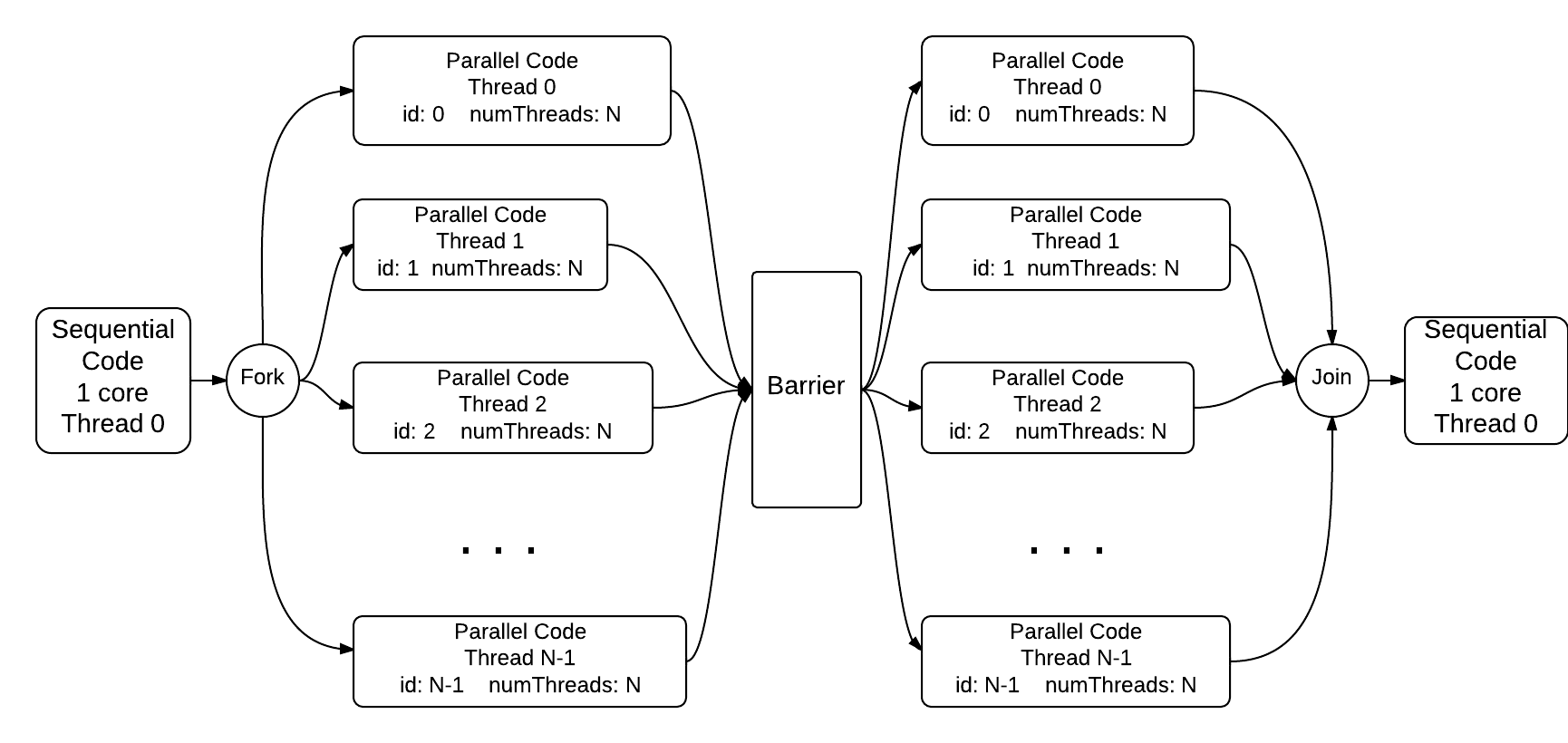
*Execute on the command line inside 04.barrier directory:*

./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();

printf("Thread %d of %d is BEFORE the barrier.**\n**", id, numThreads);

|  |  |  |
| --- | --- | --- |
| 29 |  |  |
| 30 |  |
| 31 | *//* |
| 32 |  |
| 33 |  |
| 34 |  | } |
| 35 |  |  |

*#pragma omp barrier*

printf("Thread %d of %d is AFTER the barrier.**\n**", id, numThreads);

36 printf("**\n**");

37 **return** 0;

38 }

1. **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*

# %d of %d threads**\n**",

|  |  |  |  |
| --- | --- | --- | --- |
| 33 |  | printf("Greetings from a worker, | |
| 34 |  |  | id, numThreads); |
| 35 |  | } |  |
| 36 | } |  |  |
| 37 |  |  |  |

|  |  |  |
| --- | --- | --- |
| 38 |  | printf("**\n**"); |
| 39 |  |  |
| 40 |  | **return** 0; |
| 41 | } |  |

* + 1. **Data Decomposition Algorithm Strategies and Related Coordination Strate- gies**

1. **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++) {

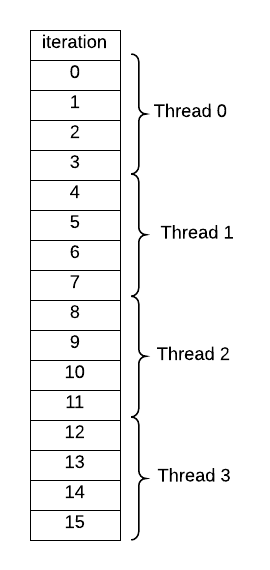
29 **int** id = omp\_get\_thread\_num();

30 printf("Thread %d performed iteration %d**\n**",

d, i);

|  |  |  |
| --- | --- | --- |
| 31 |  | 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.

1. **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) {*

*d performed iteration %d\n",*

|  |  |  |  |
| --- | --- | --- | --- |
| 46 |  | *printf("Thread %* | |
| 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.

1. **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.*

10 *\* - Uncomment #pragma in function parallelSum(),*

11 *\* but leave its reduction clause commented out*

12 *\* - Recompile and rerun. Note that correct output is NOT produced.*

13 *\* - Uncomment ’reduction(+:sum)’ clause of #pragma in parallelSum()*

14 *\* - Recompile and rerun. Note that correct output is produced again.*

15 *\*/*

16

17 *#include <stdio.h> // printf()*

18 *#include <omp.h> // OpenMP*

19 *#include <stdlib.h> // rand()*

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 **if** (argc > 1) {

31 omp\_set\_num\_threads( atoi(argv[1]) );

32 }

33

34 initialize(array, SIZE);

35 printf("**\n**Seq. sum: **\t**%d**\n**Par. sum: **\t**%d**\n\n**",

36 sequentialSum(array, SIZE),

37 parallelSum(array, SIZE) );

38

39 **return** 0;

40 }

41

42 */\* fill array with random values \*/*

43 **void** initialize(**int**\* a, **int** n) {

44 **int** i;

45 **for** (i = 0; i < n; i++) {

46 a[i] = rand() % 1000;

47 }

48 }

49

50 */\* sum the array sequentially \*/*

51 **int** sequentialSum(**int**\* a, **int** n) {

52 **int** sum = 0;

53 **int** i;

|  |  |  |  |
| --- | --- | --- | --- |
| 54 |  | **for** (i | = 0; i < n; i++) { |
| 55 |  | sum | += a[i]; |
| 56 |  | } |  |
| 57 |  | **return** | sum; |
| 58 | } |  |  |
| 59 |  |  |  |

60 */\* sum the array using multiple threads \*/*

61 **int** parallelSum(**int**\* a, **int** n) {

62 **int** sum = 0;

63 **int** i;

64 *// #pragma omp parallel for // reduction(+:sum)*

n; i++) {

|  |  |  |
| --- | --- | --- |
| 65 | **for** (i = 0; i < | |
| 66 |  | sum += a[i]; |
| 67 | } |  |

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.

1. **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 << "**\n**Usage: ./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("9332621544394415268169923885")

71 + "6266700490715968264381621468"

72 + "5929638952175999932299156089"

73 + "4146397615651828625369792082"

74 + "7223758251185210916864000000"

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.

1. **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.

1 */\* dynamicScheduling.cpp computes a table of factorial values,*

2 *\* using Owen Astrachan’s BigInt class to explore*

3 *\* OpenMP’s schedule() clause.*

4 *\**

5 *\* @author: Joel Adams, Calvin College, Dec 2015.*

6 *\**

7 *\* Usage: ./dynamicScheduling [numThreads]*

8 *\**

9 *\* Exercise:*

10 *\* - Build and run, record sequential run time in a spreadsheet*

11 *\* - Uncomment #pragma omp parallel for, rebuild,*

12 *\* run using 2, 4, 6, 8, ... threads, record run times.*

13 *\* - Uncomment schedule(dynamic), rebuild,*

14 *\* run using 2, 4, 6, 8, ... threads, record run times.*

15 *\* - Create a line chart plotting run times vs # of threads.*

16 *\*/*

17

18 *#include "BigInt.h" // class BigInt*

19 *#include <omp.h> // OpenMP functions*

20 *#include <cassert> // assert()*

21

22 */\* factorial(n) computes n!*

23 *\* @param: n, an unsigned int.*

24 *\* @return: n!, a BigInt.*

25 *\*/*

26 BigInt factorial(**unsigned** n) {

27 BigInt result = 1; *// 0! or 1!*

28

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

30 result \*= i;

31 }

32

33 **return** result;

34 }

35

36 **int** main(**int** argc, **char**\*\* argv) { *// on a 2 GHz i7 CPU:*

37 **const unsigned** MAX = 512; *// ~14 secs sequentially*

38 *// const unsigned MAX = 800; // ~60 secs sequentially*

39 BigInt factorialTable[MAX+1];

40

41 **if** (argc > 1) { omp\_set\_num\_threads( atoi(argv[1]) ); }

42

43 cout << "**\n**Depending on the speed of your computer,"

44 << "**\n** this program may take a while to complete,"

45 << "**\n** so please wait patiently...**\n**" << endl;

46

47 **double** startTime = omp\_get\_wtime();

48 *// #pragma omp parallel for // schedule(dynamic)*

49 **for** (**unsigned** i = 0; i <= MAX; i++) {

50 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("9332621544394415268169923885") |
| 66 |  | + "6266700490715968264381621468" |
| 67 |  | + "5929638952175999932299156089" |
| 68 |  | + "4146397615651828625369792082" |
| 69 |  | + "7223758251185210916864000000" |
| 70 |  | + "000000000000000000" ) ); |
| 71 |  | cout << "All tests passed!**\n**" << endl; |
| 72 | } |  |

* + 1. **Patterns used when threads share data values**

1. **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

14 *#include <stdio.h>*

15 *#include <omp.h>*

16 *#include <stdlib.h>*

17

18 *#define SIZE 100*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 19 |  | | | |
| 20 | **int** | main(**int** argc, **char**\*\* argv) { | |  |
| 21 |  | **int** i, j, ok = 1; | |  |
| 22 |  | **int** m[SIZE][SIZE]; | |  |
| 23 |  |  | |  |
| 24 |  | printf("**\n**"); | |  |
| 25 |  | *// set all array entries to 1* | |  |
| 26 | *//* | *#pragma omp parallel for* | | *// A* |
| 27 | *//* | *#pragma omp parallel for private(i,j)* | | *// B* |
| 28 |  | **for** (i = 0; i < SIZE; i++) { | |  |
| 29 |  | **for** (j = 0; j < SIZE; j++) { | |  |
| 30 |  |  | m[i][j] = 1; | |
| 31 |  |  | } | |
| 32 |  | } |  | |
| 33 |  |  |  | |

34 *// test (without using threads)*

35 **for** (i = 0; i < SIZE; i++) {

36 **for** (j = 0; j < SIZE; j++) {

37 **if** ( m[i][j] != 1 ) {

38 printf("Element [%d,%d] not set... **\n**", i, j);

39 ok = 0;

40 }

41 }

42 }

43

44 **if** ( ok ) {

f("**\n**All elements correctly set to 1**\n\n**");

|  |  |  |
| --- | --- | --- |
| 45 |  | print |
| 46 |  | } |
| 47 |  |  |
| 48 |  | **return** 0; |
| 49 | } |  |

1. **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.

1 */\* atomic.c*

2 *\* ... illustrates a race condition when multiple threads read from /*

3 *\* write to a shared variable (and explores OpenMP atomic operations).*

4 *\**

5 *\* Joel Adams, Calvin College, November 2009.*

6 *\**

7 *\* Usage: ./atomic*

8 *\**

9 *\* Exercise:*

10 *\* - Compile and run 10 times; note that it always produces the correct balance: $1,000,000.00*

11 *\* - To parallelize, uncomment A, recompile and rerun multiple times, compare results*

12 *\* - To fix: uncomment B, recompile and rerun, compare*

13 *\*/*

14

15 *#include <stdio.h> // printf()*

16 *#include <omp.h> // OpenMP*

17

18 **int** main() {

19 **const int** REPS = 1000000;

20 **int** i;

21 **double** balance = 0.0;

22

23 printf("**\n**Your starting bank account balance is %0.2f**\n**",

24 balance);

25

26 *// simulate many deposits*

27 *// #pragma omp parallel for // A*

28 **for** (i = 0; i < REPS; i++) {

29 *// #pragma omp atomic // B*

30 balance += 1.0;

31 }

32

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

|  |  |  |
| --- | --- | --- |
| 34 |  |  |
| 35 |  |  |
| 36 |  | **return** 0; |
| 37 | } |  |

1. **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.

1 */\* critical.c*

2 *\* ... fixes a race condition when multiple threads read from /*

3 *\* write to a shared variable using the OpenMP critical directive.*

4 *\**

5 *\* Joel Adams, Calvin College, November 2009.*

6 *\**

7 *\* Usage: ./critical*

8 *\**

9 *\* Exercise:*

10 *\* - Compile and run several times; note that it always produces the correct balance $1,000,000.00*

11 *\* - Comment out A; recompile/run, and note incorrect result*

12 *\* - To fix: uncomment B1+B2+B3, recompile and rerun, compare*

13 *\*/*

14

15 *#include<stdio.h>*

16 *#include<omp.h>*

17

18 **int** main() {

19 **const int** REPS = 1000000;

20 **int** i;

21 **double** balance = 0.0;

22

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

24

25 *// simulate many deposits*

26 *#pragma omp parallel for*

27 **for** (i = 0; i < REPS; i++) {

28 *#pragma omp atomic // A*

29 *// #pragma omp critical // B1*

30 *// { // B2*

31 balance += 1.0;

32 *// } // B3*

33 }

34

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

|  |  |  |
| --- | --- | --- |
| 36 |  |  |
| 37 |  |  |
| 38 |  | **return** 0; |
| 39 | } |  |

1. **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.

1 */\* critical2.c*

2 *\* ... compares the performance of OpenMP’s critical and atomic directives*

3 *\**

4 *\* Joel Adams, Calvin College, November 2009.*

5 *\**

6 *\* Usage: ./critical2*

7 *\**

8 *\* Exercise:*

9 *\* - Compile, run, compare times for critical vs. atomic*

10 *\* - Note how much more costly critical is than atomic*

11 *\* - Research: Create an expression that, when assigned to balance,*

12 *\* critical can handle but atomic cannot*

13 *\*/*

14

15 *#include<stdio.h>*

16 *#include<omp.h>*

17

18 **void** print(**char**\* label, **int** reps, **double** balance, **double** total, **double** average) {

19 printf("**\n**After %d $1 deposits using ’%s’: \

20 **\n\t**- balance = %0.2f, \

21 **\n\t**- total time = %0.12f, \

22 **\n\t**- average time per deposit = %0.12f**\n\n**",

23 reps, label, balance, total, average);

24 }

25

26 **int** main() {

27 **const int** REPS = 1000000;

28 **int** i;

29 **double** balance = 0.0,

30 startTime = 0.0,

31 stopTime = 0.0,

32 atomicTime = 0.0,

33 criticalTime = 0.0;

34

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

36

37 *// simulate many deposits using atomic*

38 startTime = omp\_get\_wtime();

39 *#pragma omp parallel for*

40 **for** (i = 0; i < REPS; i++) {

41 *#pragma omp atomic*

42 balance += 1.0;

43 }

44 stopTime = omp\_get\_wtime();

45 atomicTime = stopTime - startTime;

46 print("atomic", REPS, balance, atomicTime, atomicTime/REPS);

47

48

49 *// simulate the same number of deposits using critical*

50 balance = 0.0;

51 startTime = omp\_get\_wtime();

52 *#pragma omp parallel for*

53 **for** (i = 0; i < REPS; i++) {

54 *#pragma omp critical*

55 {

56 balance += 1.0;

57 }

58 }

59 stopTime = omp\_get\_wtime();

60 criticalTime = stopTime - startTime;

61 print("critical", REPS, balance, criticalTime, criticalTime/REPS);

62

63 printf("criticalTime / atomicTime ratio: %0.12f**\n\n**",

riticalTime

|  |  |  |
| --- | --- | --- |
| 64 |  | c |
| 65 |  |  |
| 66 |  | **return** 0; |
| 67 | } |  |

/ atomicTime);

1. **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.

1 */\* critical3.c*

2 *\* ... a simple case where OpenMP’s critical works but atomic does not.*

3 *\**

4 *\* Joel Adams, Calvin College, November 2009.*

5 *\**

6 *\* Usage: ./critical3*

7 *\**

8 *\* Exercise:*

9 *\* - Compile, run, note resulting output is correct.*

10 *\* - Uncomment line A, recompile, rerun, note results.*

11 *\* - Uncomment line B, recompile, note results.*

12 *\* - Recomment line B, uncomment line C, recompile,*

13 *\* rerun, note change in results.*

14 *\*/*

15

16 *#include<iostream> // cout*

17 *#include<omp.h> // openmp*

18 **using namespace** std;

19

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

21 cout << "**\n**";

22

23 **if** (argc > 1) {

24 omp\_set\_num\_threads( atoi(argv[1]) );

25 }

26

27 *// #pragma omp parallel // A*

28 {

29 **int** id = omp\_get\_thread\_num();

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 30 |  |  | **int** numThreads = omp\_get\_num\_threads(); |  | |
| 31 | *//* |  | *#pragma omp atomic* | *//* | *B* |
| 32 | *//* |  | *#pragma omp critical* | *//* | *C* |
| 33 |  |  | cout << "Hello from thread #" << id |  |  |
| 34 |  |  | << " out of " << numThreads |  |  |
| 35 |  |  | << " threads.**\n**"; |  |  |
| 36 |  | } |  |  |  |
| 37 |  |  |  |  |  |

38 cout << "**\n**";

39 }

**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.

* + 1. **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.

1. **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>*

13 *#include <omp.h>*

14 *#include <stdlib.h>*

15

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

17

18 printf("**\n**Before...**\n\n**");

19

20 *#pragma omp parallel sections num\_threads(4)*

21 {

22 *#pragma omp section*

23 {

24 printf("Task/section A performed by thread %d**\n**",

25 omp\_get\_thread\_num() );

26 }

27 *#pragma omp section*

28 {

29 printf("Task/section B performed by thread %d**\n**",

30 omp\_get\_thread\_num() );

31 }

32 *#pragma omp section*

33 {

34 printf("Task/section C performed by thread %d**\n**",

35 omp\_get\_thread\_num() );

36 }

37 *#pragma omp section*

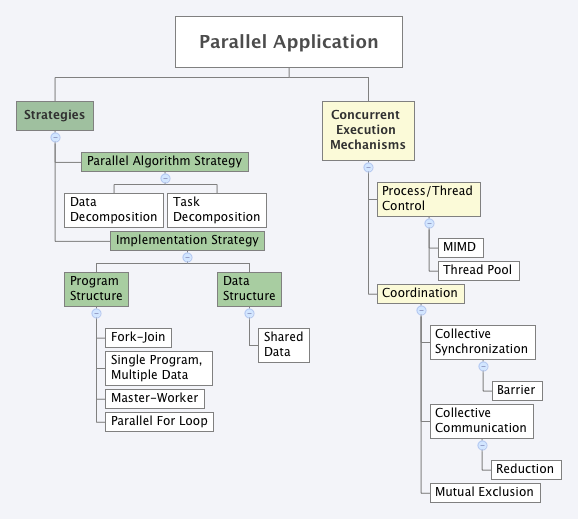
38 {

by thread %d**\n**",

|  |  |  |
| --- | --- | --- |
| 39 |  | printf("Task/section D performed |
| 40 |  | omp\_get\_thread\_num() ); |
| 41 |  | } |
| 42 |  | } |
| 43 |  |  |
| 44 |  | printf("**\n**After...**\n\n**"); |
| 45 |  |  |
| 46 |  | **return** 0; |
| 47 | } |  |

* + 1. **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.