Q.1 Write a C/MPI program Gather sets of 30 integers from each process in the group to the root. [4]

14.8.3 Gather (MPI_GATHER)

Messages sent by the group processes are concatenated in rank order, and the resulting message is received by the root process as if by a call to MPI_RECV. General and derived data types are allowed for both send and receive as shown in the following example:

```
/* MPI_Gather Example: Gather an array of 100 integers */
/* from multiple processes */

/* number of active processes */
int gsize;

/* send buffer */
int sendarray[100];

/* root process */
int root = 0;

/* current process number */
int myrank;

/* receive buffer */
int *rbuf;

MPI_Comm_rank(MPI_COMM_WORLD, &myrank);




  if ( myrank == root)
  {
      MPI_Comm_size ( comm, &gsize);
      rbuf = (int *) malloc (gsize * 100 * sizeof(int) );
  }

  MPI_Gather( sendarray, 100, MPI_INT
              , rbuf, 100, MPI_INT
              , root, MPI_COMM_WORLD);
```

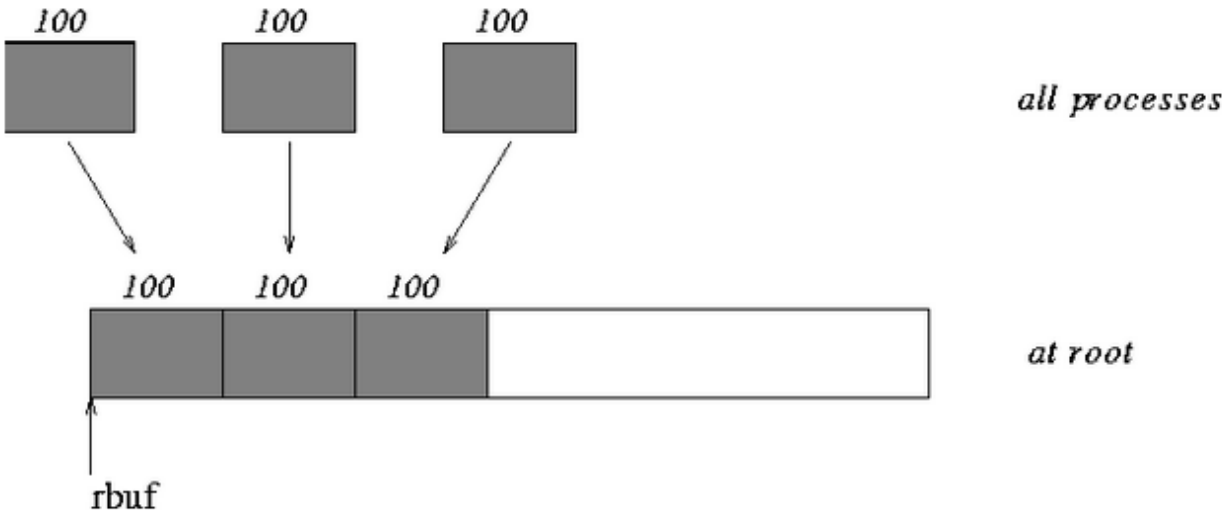


Figure 3: The root process gathers 100 `ints` from each process in the group.

http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70

Q.2 What are condition variables? How and when do you use in the OpenMP program? [4]

Pthread Routines

Pthreads API can be informally grouped into three major classes:

Thread management: The first class of functions work directly on threads - creating, detaching, joining, etc. They include functions to set/query thread attributes (joinable, scheduling etc.)

Mutexes: The second class of functions deal with synchronization, called a "mutex", which is an abbreviation for "mutual exclusion". Mutex functions provided for creating, destroying, locking and unlocking mutexes. They are also supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Pthread Routines

Condition variables: The third class of functions address communications between threads that share a mutex. They are based upon programmer specified conditions.

This class includes functions to create, destroy, wait and signal based upon specified variable values. Functions to set/query condition variable attributes are also included.

Condition Variables

- In a critical section (i.e. where a mutex has been used), a thread can suspend itself on a *condition variable* if the state of the computation is not right for it to proceed.
  - It will suspend by *waiting* on a condition variable.
  - It will, however, release the critical section lock (mutex) .
  - When that condition variable is *signaled,* it will become ready again; it will attempt to reacquire that critical section lock and only then will be able proceed.
- With Posix threads, a condition variable can be associated with only one mutex variable!

Condition Variables

Often, a critical section is to be executed if a specific global condition exists; for example, if a certain value of a variable has been reached.

With locks, the global variable would need to be examined at frequent intervals ("polled") within a critical section.

This is a very time-consuming and unproductive exercise.

Can be overcome by introducing so-called *condition variable*

Condition Variables

Main Thread

- o Declare and initialize global data/variables which require synchronization (such as "count")
- o Declare and initialize a condition variable object
- o Declare and initialize an associated mutex
- o Create threads A and B to do work

Thread A

- Do work up to the point where a certain condition must occur (such as "count" must reach a specified value)
- Lock associated mutex and check value of a global variable
- Call pthread_cond_wait() to perform a blocking wait for signal from Thread-B. *Note that a call to pthread_cond_wait() automatically and atomically unlocks the associated mutex variable so that it can be used by Thread-B.*
- When signalled, wake up. Mutex is automatically and atomically locked.
- Explicitly unlock mutex
- Continue

Thread B

- Do work
- Lock associated mutex
- Change the value of the global variable that Thread-A is waiting upon.
- Check value of the global Thread-A wait variable. If it fulfills the desired condition, signal Thread-A.
- Unlock mutex.
- Continue

Main Thread
Join / Continue

Pthread Condition Variables

Associated with a specific mutex. Given declarations:

```
pthread_cond_t cond1;
pthread_mutex_t mutex1;
pthread_cond_init(&cond1, NULL);
pthread_mutex_init(&mutex1, NULL);
```

the Pthreads arrangement for signal and wait is as follows:

```
action()                                counter()
{                                         {
          .                                    .
          .                                    .
pthread_mutex_lock(&mutex1);            pthread_mutex_lock(&mutex1);
while (c <> 0)                          c--;
  pthread_cond_wait(cond1,mutex1);      if (c == 0) pthread_cond_signal(cond1);
pthread_mutex_unlock(&mutex1);          pthread_mutex_unlock(&mutex1);
take_action();                            .
          .                               .
          .                               .
}                                       }
```

Signals are *not* remembered - threads must already be waiting for a signal to receive it.

Condition Variables

- Allows for threads to wait for a condition to be satisfied
- Used along with a mutex lock
- pthread_cond_wait puts a thread to sleep waiting for a pthread_cond_signal to be issued by another thread
- Example: producer-consumers interaction, wake up a consumer when there is a task available

```
pthread_mutex_lock(&mut);
while (tasks_left == 0) {
    pthread_cond_wait(&cv, &mut);
}
pthread_mutex_unlock(&mut);
```

```
pthread_mutex_lock(&mut);
if (tasks_left != 0) {
    pthread_cond_signal(&cv, &mut);
}
pthread_mutex_unlock(&mut);
```

Condition Variable Issues

- Programming discipline: always obtain the lock before signaling or waiting on a condition variable
- Makes sure that no signals are lost
- pthread_cond_wait implicitly relinquishes the lock and obtains it back when woken up
- pthread_cond_signal wakes up exactly one waiting thread, pthread_cond_broadcast wakes up all waiting threads
- signal could be used without locking
- A thread could wake up and reset the program level criteria; broadcast does not imply all threads are runnable

Q.5 What are the alternatives for programming shared memory multiprocessors. Which one is better and why? [4]

Shared Memory Programming

Several Thread Libraries/systems

- PTHREADS is the POSIX Standard
  - Relatively low level
  - Portable but possibly slow; relatively heavyweight
- OpenMP standard for application level programming
  - Support for scientific programming on shared memory
  - http://www.openMP.org
- TBB: Thread Building Blocks
  - Intel
- CILK: Language of the C “ilk”
  - Lightweight threads embedded into C
- Java threads
  - Built on top of POSIX threads
  - Object within Java language

There are several alternatives for programming shared memory multiprocessor systems:

- Using a completely new programming language for parallel programming
- Modifying the syntax of an existing sequential programming language to create a parallel programing language
- Using an existing sequential programming language supplemented with compiler directives for specifying parallelism
- Using library routines with an existing sequential programming language
- Using heavyweight processes
- Using threads

One could also use a regular sequential programming language and ask a *parallelizing compiler* to convert the sequential program into parallel executable code. In that case, the compiler establishes which statements can be executed simultaneously. It might rearrange the statements to achieve concurrency, but the original intent of the programmer must be left intact. This method was investigated extensively in the 1970s. Using a completely new parallel programming language is of very limited appeal, for it requires one to learn a new language from scratch. Only one example of this has been used to any extent, the Ada language promoted by the U.S. Department of Defense.

Taking an existing sequential language and modifying it is more attractive, because then one only needs to learn the modifications. The most appealing way of doing this is to use compiler directives and library routines rather than modifying the syntax. An accepted standard for doing this is OpenMP. A special compiler is still needed.

Interestingly, Stroustrup, the inventor of C++, in the preface to Wilson and Lu (1996), says that he did not include any concurrency features in the original C++ specification though he could have done so. His conclusion was that "no single model of concurrency would serve more than a small fraction of the user community well." He also has a "weakness for the library approaches because these offer a higher degree of portability than approaches based upon language extensions."

In this chapter, we will start with traditional processes and then introduce the *thread* using the thread Pthreads standard. Pthreads is readily available on a multitude of platforms (single workstations and multiprocessor systems). Java also provides thread-based capabilities and offers some high-level features that are described here. It is perfectly feasible to use Java for thread-based parallel programming if an implementation is provided for the target multiprocessor system.

Which is better?

Pthreads and OpenMP represent two totally different multiprocessing paradigms.

Pthreads is a very low-level API for working with threads. Thus, you have extremely fine-grained control over thread management (create/join/etc), mutexes, and so on. It's fairly bare-bones.

On the other hand, OpenMP is *much* higher level, is more portable and doesn't limit you to using C. It's also much more easily scaled than pthreads. One specific example of this is OpenMP's work-sharing constructs, which let you divide work across multiple threads with relative ease. (See also Wikipedia's pros and cons list.)

That said, you've really provided no detail about the specific program you're implementing, or how you plan on using it, so it's fairly impossible to recommend one API over the other.

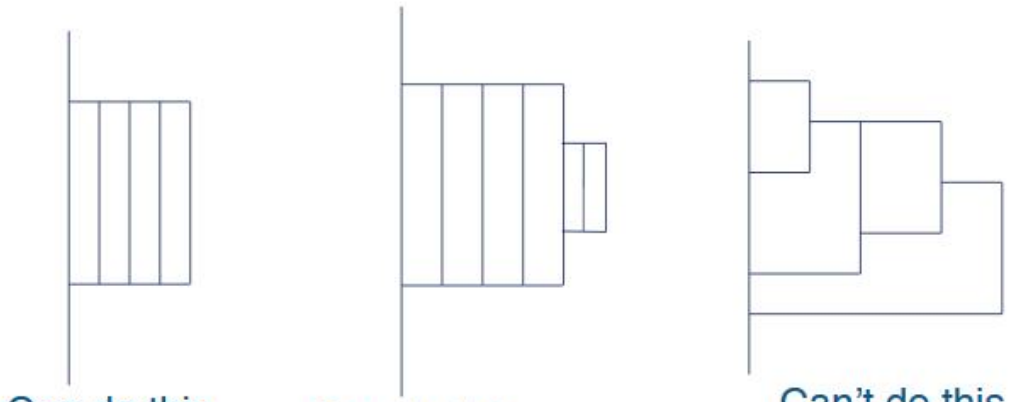
It basically boils down to what level of control you want over your parallelization. OpenMP is great if all you want to do is add a few #pragma statements and have a parallel version of your code quite quickly. If you want to do really interesting things with MIMD coding or complex queueing, you can still do all this with OpenMP, but it is probably a lot more straightforward to use threading in that case. OpenMP also has similar advantages in portability in that a lot of compilers for different platforms support it now, as with pthreads.

So you're absolutely correct - if you need fine-tuned control over your parallelization, use pthreads. If you want to parallelize with as little work as possible, use OpenMP.

What's wrong with OpenMP?

- OpenMP is designed for programs where you want a fixed number of threads, and you always want the threads to be consuming CPU cycles.
  - cannot arbitrarily start/stop threads
  - cannot put threads to sleep and wake them up later
- OpenMP is good for programs where each thread is doing (more-or-less) the same thing.
- Although OpenMP supports C++, it's not especially OO friendly
  - though it is gradually getting better.
- OpenMP doesn't support Java
- OpenMP programs don't run on distributed memory architectures (e.g. clusters of PCs)

What's wrong with OpenMP? (cont.)



What are the alternatives?

- Threading libraries
  - POSIX threads
  - Boost threads
  - Intel TBB
  - Java threads

- Other programming models
  - MPI
  - PGAS languages
  - Novelties

Threading libraries

- Pure library interfaces (no directives)
- Lower level of abstraction than OpenMP
  - requires more changes to code
  - harder to maintain
  - some are less portable than OpenMP
- Free software

POSIX threads

- POSIX threads (or Pthreads) is a standard library for shared memory programming without directives.
  - Part of the ANSI/IEEE 1003.1 standard (1996)
- Interface is a C library
  - no standard Fortran interface
  - can be used with C++, but not OO friendly
- Widely available
  - even for Windows
  - typically installed as part of OS
  - code is pretty portable
- Lots of low-level control over behaviour of threads

Intel Thread Building Blocks (TBB)

- C++ library for multithreaded programming
- Offers somewhat higher level of abstraction that POSIX/ BOOST threads
  - notion of tasks rather that explicit threads
  - support for parallel loops and reductions
  - support for concurrency on containers
- Moderately portable
  - support for Intel and gcc compilers on Linux and Mac OS X, Intel and Visual C++ on Windows
  - no build required to install

Java threads

- Threads are an inbuilt part of the Java language
- Very portable (available in every Java VM)
- Java has lots of nice properties as a programming language
  - high performance isn't necessarily one of them!
- Well integrated into Java's OO model
- Both explicit thread creation and task models
- Synchronisation methods
  - every object contains a mutex lock
  - condition variables, barriers, explicit lock objects

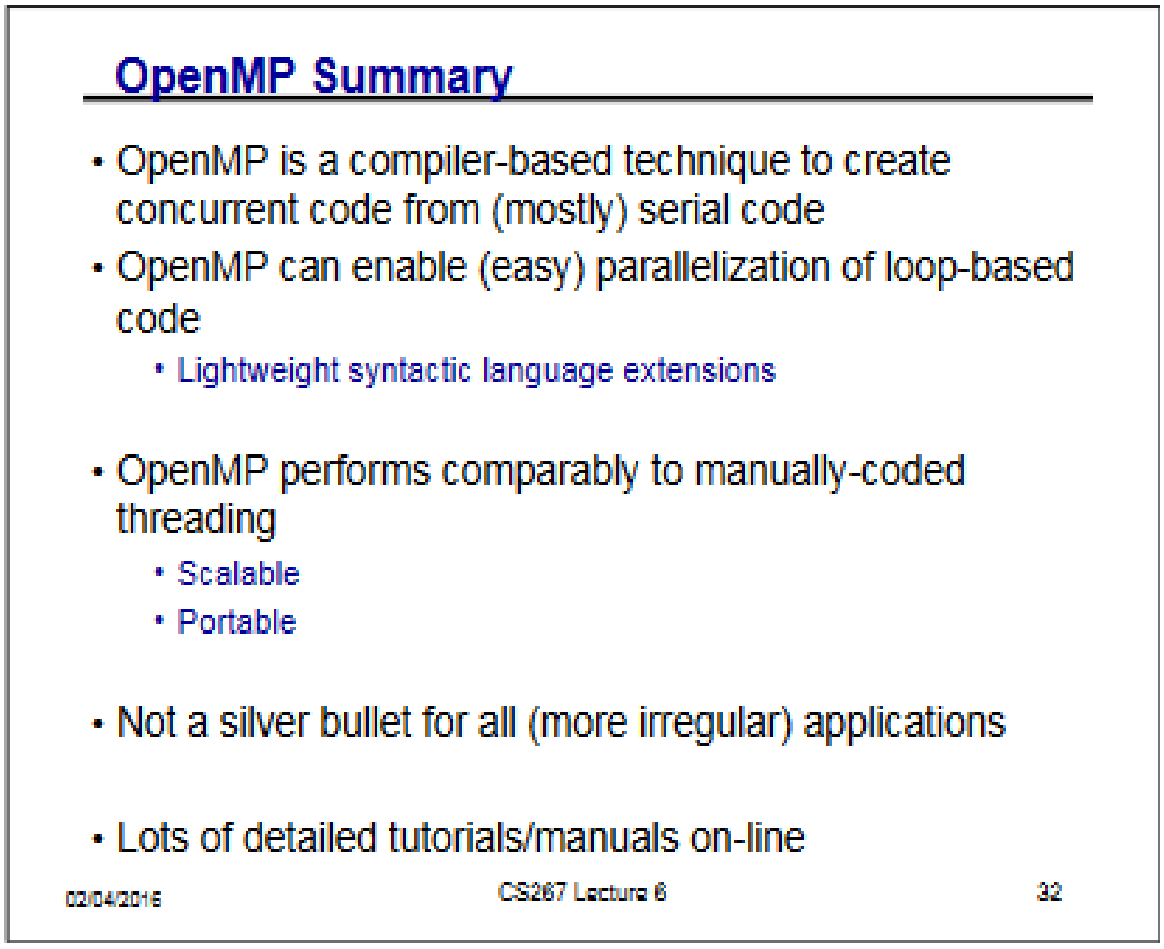
- `http://java.sun.com/docs/books/tutorial/essential/concurrency/`

MPI

- Message Passing Interface (MPI)
- Principal method used for programming distributed memory architectures
  - library interface for Fortran, C, C++
- MPI programs will run just fine on multicore systems
- Need to install an MPI library and configure it to use shared memory to pass messages
- MPI programs typically take much longer to develop than OpenMP programs (months not weeks)
  - extremely portable and scalable solution
- Hybrid OpenMP/MPI becoming a common solution for very large systems

What to do?

- OpenMP is a good solution for many programs in the scientific computing area.
- However, there may be very sound reasons for not using OpenMP.
- If you are going to use one of the alternatives, you should convince yourself carefully of why you are not using OpenMP!



Which Threading Model is Right For You?

OpenMP is convenient because it does not lock the software into a preset number of threads. This kind of lock-in poses a big problem for threaded applications that use lower-level APIs such as Pthreads or Win32. How can the software written with those APIs scale the number of threads when running on a platform where more processors are available? One approach has been to use threading pools, in which a bunch of threads are created at program start up and the work is distributed among them. However, this approach requires considerable thread-specific code and there is no guarantee that it will scale optimally with the number of available processors. Wi th OpenMP, the number need not be specified.

OpenMP's pragmas have another key advantage: by disabling support for OpenMP, the code can be compiled as a single-threaded application. Compiling the code this way can be tremendously advantageous when debugging a program. Without this option, developers will frequently find it difficult to tell whether complex code is working incorrectly because of a threading problem or because of a design error unrelated to threading.

Should developers need finer-grained control, they can avail themselves of OpenMP's threading

API. It includes a small set of functions that fall into three areas: querying the execution environment's threading resources and setting the current number of threads; setting, managing, and releasing locks to resolve resource access between threads; and a small timing interface. Use of this API is discouraged because it takes away the benefits provided by the pragma-only approach. At this level, the OpenMP API is a small subset of the functionality offered by Pthreads. Both APIs are portable, but Pthreads offers a much greater range of primitive functions that provide finer-grained control over threading operations. So, in applications in which threads have to be individually managed, Pthreads or the native threading API (such as Win32 on Windows) would be the more natural choice.

To run OpenMP, a developer must have a compiler that supports the standard. On Linux and Windows, Intel® Compilers for C/C++ and Fortran support OpenMP. On the UNIX platform, SGI, Sun, HP, and IBM all provide OpenMP-compliant compilers. Open-source OpenMP compilers can be found at http://openmp.org/wp/openmp-compilers/.

So, if you're writing UNIX or Linux applications for HPC, look at both Pthreads and OpenMP. You might well find OpenMP to be an elegant solution

Why should we use OpenMP over pthreads?

Both of them use threads to divide the work, right? Well, OpenMP enables you to specify a program at a higher abstraction than pthreads. You need not explicitly write the synchronization part of the code which is the trickiest part in multi-threaded computing.

1. http://www.cs.nyu.edu/courses/fall10/G22.2945-001/slides/lect3-4.pdf

Extra

OpenMP is a way to program on shared memory devices. This means that the parallelism occurs where every parallel thread has access to all of your data.

You can think of it as: parallelism can happen during execution of a specific `for` loop by splitting up the loop among the different threads.

MPI is a way to program on distributed memory devices. This means that the parallelism occurs where every parallel process is working in its own memory space in isolation from the others.

You can think of it as: every bit of code you've written is executed independently by every process. The parallelism occurs because you tell each process exactly which part of the global problem they should be working on based entirely on their process ID.

The way in which you write an OpenMP and MPI program, of course, is also very different.