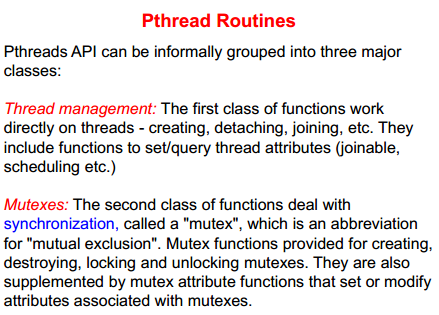
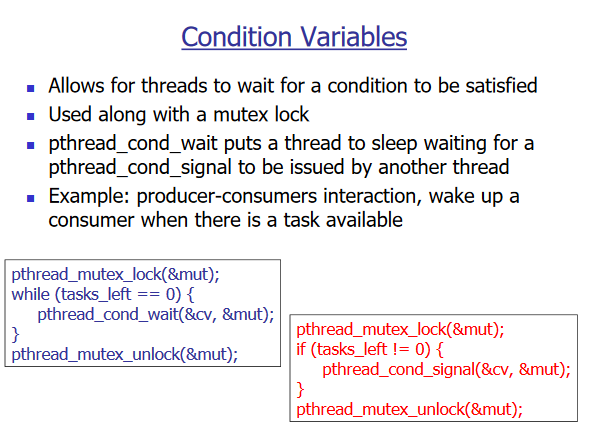
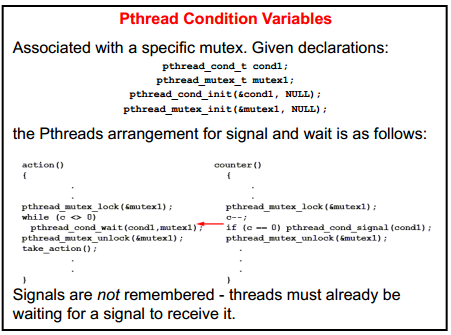
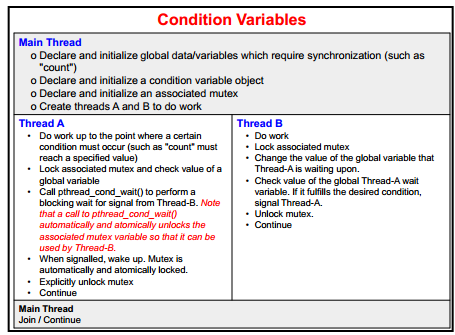
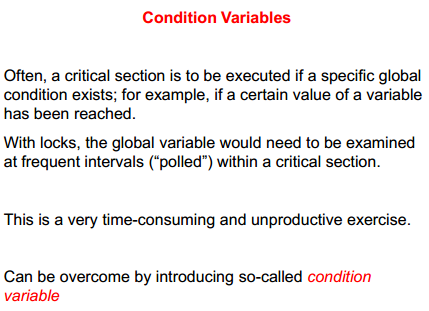
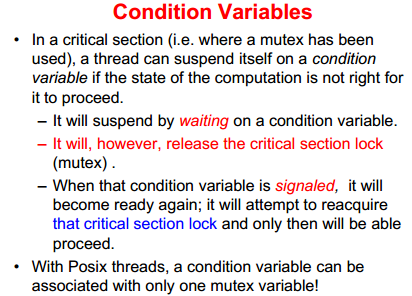
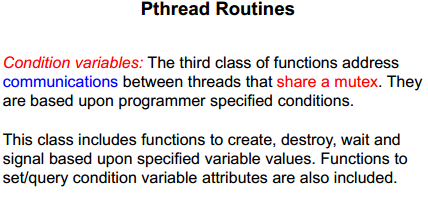
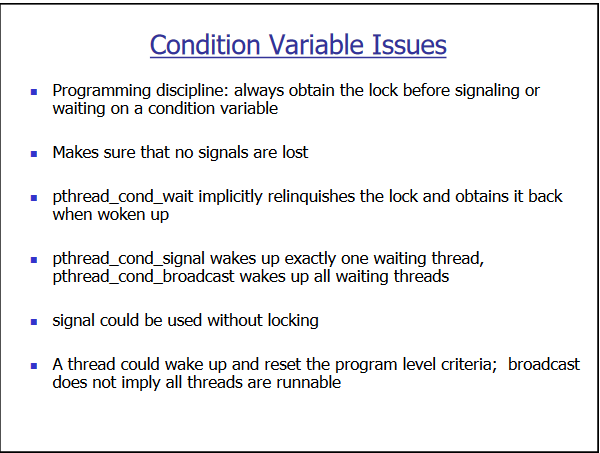


<http://mpi-forum.org/docs/mpi-1.1/mpi-11-html/node70.html#Node70>

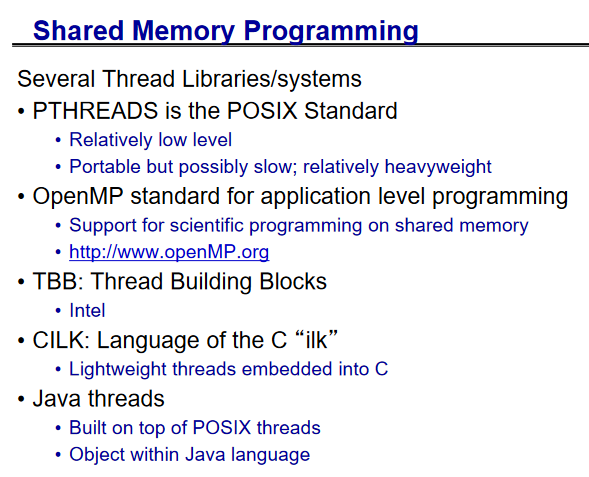


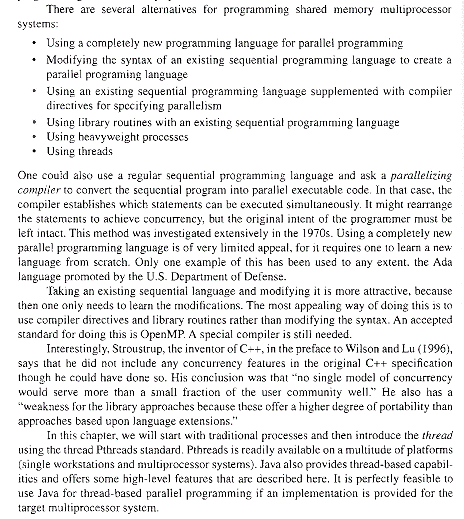












Which is better?

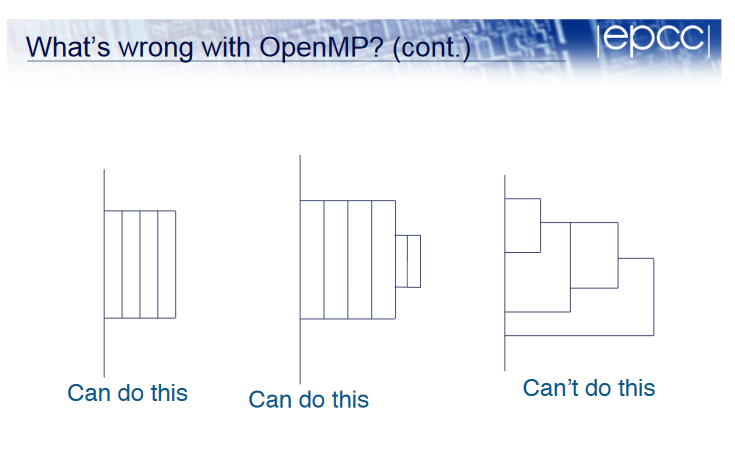
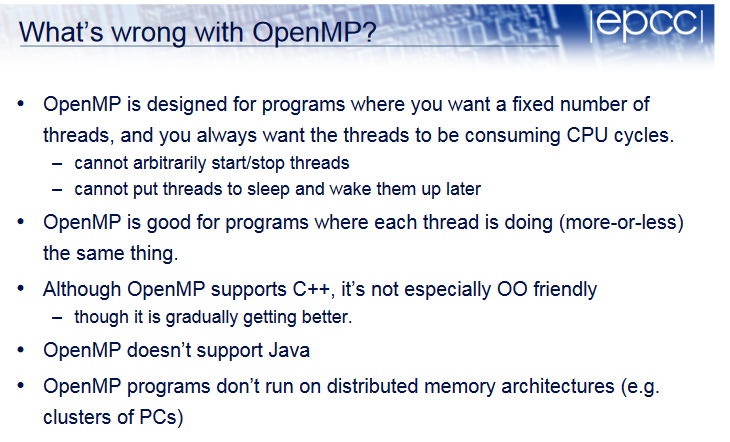
Pthreads and OpenMP represent two totally different multiprocessing paradigms.

[Pthreads](http://en.wikipedia.org/wiki/POSIX_Threads) 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](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/OpenMP#Pros_and_cons).)

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. |



### 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](https://clustercomputing.wordpress.com/2009/11/24/openmp/www.llnl.gov/computing/tutorials/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

