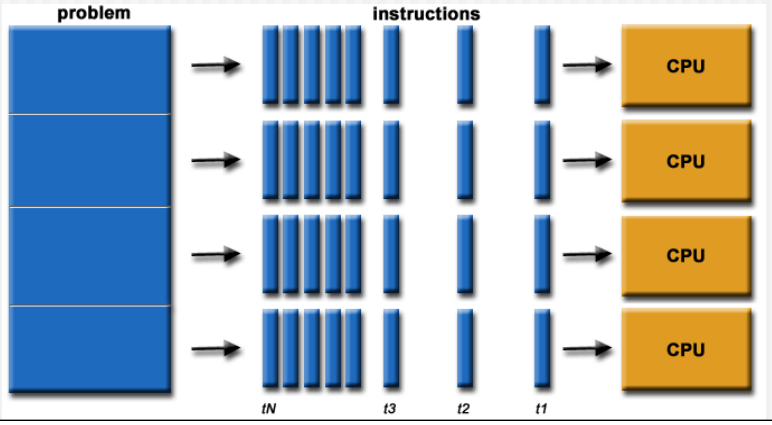
Lecture 8
Approaches to Program Parallel Machines

- Threads
- OpenMP
- Message Passing (MPI)
- HPF
- PGAS Languages

1

Problem

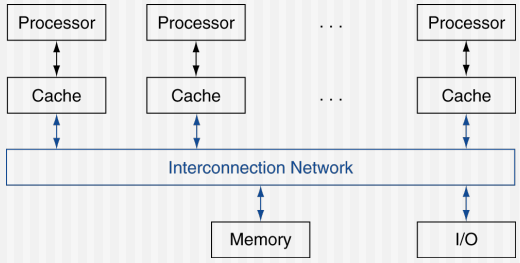
- How to instruct the computer to work on a problem in parallel
  - Efficient
  - Robust



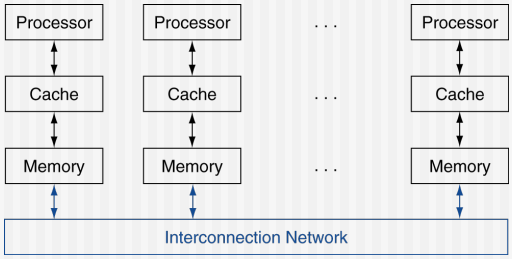
2

Different Memory Organizations

- Shared Memory



- Distributed Memory



3

Differences in Programming

- Shared Memory
  - Global Address Space
  - Need to protect access to global variables
  - Main Programming Models:
    - Threads
    - OpenMP
- Distributed Memory
  - No Global Address Space
  - Requires communication for non-local accesses
  - Programming model can still provide global address space
  - Main Programming Models:
    - Message Passing (MPI)
    - Parallel Languages (HPF, PGAS languages)

4

Programming for Shared Memory

Threads
OpenMP

5

Threads

- A thread is defined as an independent stream of instructions that can be scheduled to run as such by the operating system.
  - To the software developer, the concept of a "procedure" that runs independently from its main program may best describe a thread.
  - A multi-threaded program is a program where the main program contains a number of procedures that can potentially be scheduled independently.
- This independent flow of control is accomplished because a thread maintains its own:
  - Stack pointer
  - Registers
  - Scheduling properties (such as policy or priority)
  - Set of pending and blocked signals
  - Thread specific data.

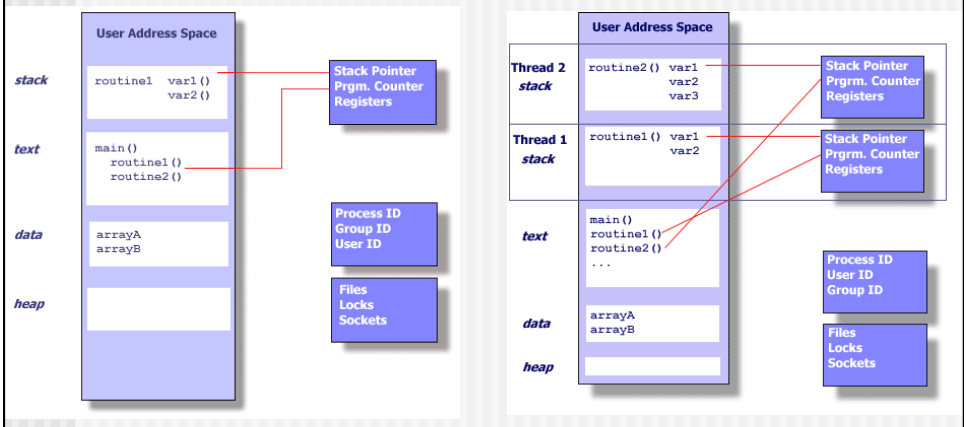
6

Threads Cont'd

- In the UNIX environment a thread:
  - Exists within a process and uses the process resources
  - Has its own independent flow of control as long as its parent process exists and the OS supports it
  - Duplicates only the essential resources it needs to be independently schedulable
  - May share the process resources with other threads that act equally independently (and dependently)
  - Dies if the parent process dies - or something similar
  - Is "lightweight" because most of the overhead has already been accomplished through the creation of its process.

7

Threads within a Process



Process

Threads

8

Pthreads

- Historically, every vendor had their own thread implementations
- In order to take full advantage of the capabilities provided by threads, a standardized programming interface was required.
- For UNIX systems, this interface has been specified by the IEEE POSIX 1003.1c standard (1995).
- Implementations adhering to this standard are referred to as POSIX threads, or Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary API's.
- You will learn programming Pthreads later in the course

9

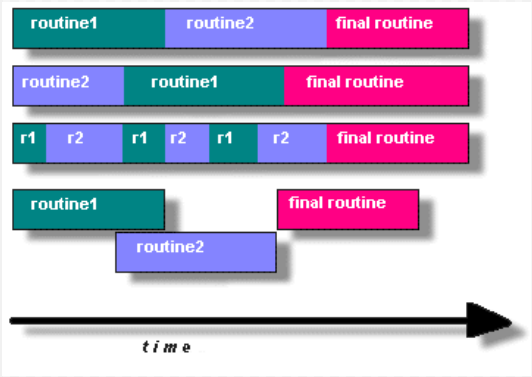
Why Threads

- Could use different processes instead?
- Threads are lightweight
  - Thread creation only takes a few percent of process creation time
- Threads share address space
  - Processes would need inter-process communication
  - Much more efficient data exchange between threads
    - E.g. Using message passing on shared memory typically involves memory copy

10

How use Threads?

- Organize program into discrete, independent tasks
  - Can use task and/or data parallelism
  - Task need to be able to execute concurrently



11

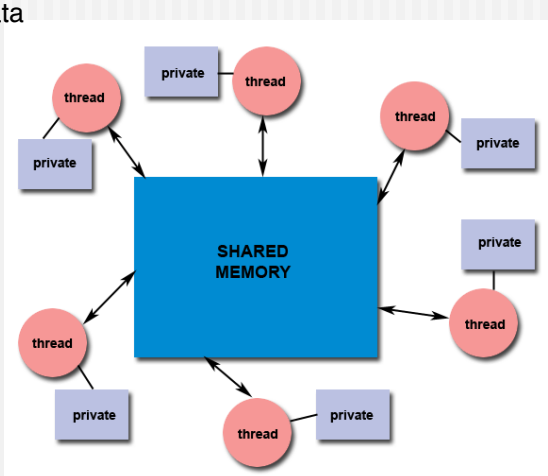
Common Models

- Master/Worker
  - A single master thread assigns work to other threads, the workers
- Pipeline
- Peer
  - Similar to master/worker but master also participates in work

12

Shared Memory Model

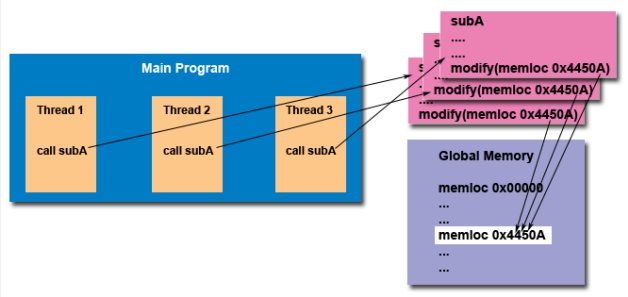
- All threads have access to the same global, shared memory
- Threads also have their own private data
- Programmers are responsible for synchronizing access globally shared data



13

Thread Safety

- Thread safety is a property of an application to execute in multiple threads simultaneously in a correct manor without creating race conditions
- If use external libraries make 100% sure they are thread safe - otherwise serialize their usage



14

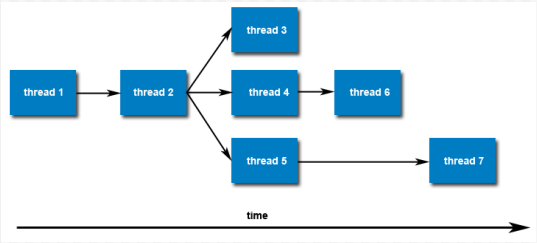
Pthreads API

- Thread management
  - Creating, detaching, joining, attribute query, …
- Synchronization
  - Mutexes
    - Creating, destroying, locking, unlocking
  - Condition variables
    - Create, destroy, wait, signal
  - Locks and barriers

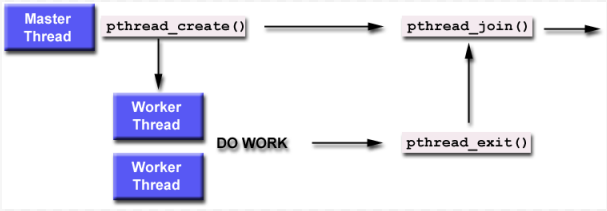
15

Thread Management

- Thread creation



- Thread exit/Thread Join



16

Synchronization

- Mutex
  - mutex_init, mutex_destroy, mutex_lock, mutex_trylock, mutex_unlock
- Condition variables
  - cond_init, cond_destroy, cond_wait, cond_signal, cond_broadcast
- Locks and barriers
  - barrier_init, barrier_destroy, barrier_wait
  - rwlock_init, rwlock_destroy, rwlock_rdlock, rwlock_wrlock, …

17

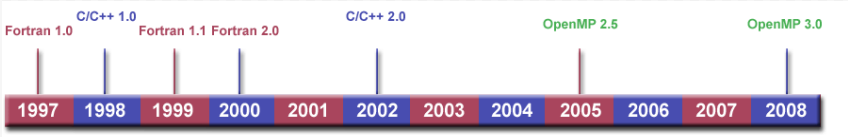
OpenMP

18

OpenMP

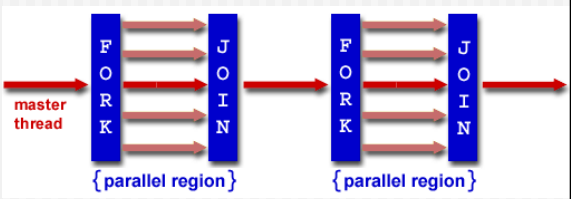
- Open Multi-Processing
- OpenMP provides a portable, scalable model for developers of shared memory parallel applications
- Jointly defined by a group of major computer hardware and software vendors

- Consists of
  - Compiler Directives
  - Runtime Library Routines
  - Environment Variables



Features of OpenMP

- **Shared Memory, Thread Based Parallelism:**
  - OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. A shared memory process consists of multiple threads.
- **Explicit Parallelism:**
  - OpenMP is an explicit (not automatic) programming model, offering the programmer full control over parallelization.
- **Fork - Join Model:**
  - OpenMP uses the fork-join model of parallel execution:



- **Compiler Directive Based:**
  - Most OpenMP parallelism is specified through the use of compiler directives which are imbedded in C/C++ or Fortran source code.

20

Features of OpenMP Cont'd

- **Nested Parallelism Support:**
  - Supports placement of parallel constructs inside of other parallel constructs
- **Dynamic Threads:**
  - Provides for dynamically altering the number of threads which may used to execute different parallel regions.
- **I/O:**
  - OpenMP specifies nothing about parallel I/O. This is particularly important if multiple threads attempt to write/read from the same file.
  - If every thread conducts I/O to a different file, the issues are not as significant.
  - It is entirely up to the programmer to insure that I/O is conducted correctly within the context of a multi-threaded program.
- **Memory Model: FLUSH Often?**
  - OpenMP provides a "relaxed-consistency" and "temporary" view of thread memory (in their words). In other words, threads can "cache" their data and are not required to maintain exact consistency with real memory all of the time.
  - When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is FLUSHed by all threads as needed.
- Not all features are mandatory

21

Caching is decided by compiler + OS

Need to think hard about how to avoid Flushing (or mutex'es)

OpenMP Code Structure

```
#include <omp.h>
main ()  {
  int var1, var2, var3;
Serial code
.
.
Beginning of parallel section.
Fork a team of threads. Specify variable scoping
#pragma omp parallel private(var1, var2) shared(var3)  {
Parallel section executed by all threads
.
.
All threads join master thread and disband
}
Resume serial code
.
.
}
```

- Details in second half of the course

22

Distributed Memory

Message Passing (MPI)

Parallel Languages (HPF, PGAS Languages)

23

Attention

- Distributed Memory programming models can often also be applied to shared memory
  - Parallel languages:
    - Runtime system based on message passing or threads
    - Compiler support
  - Message passing
    - Use shared memory to do message passing - typically involves extra copies due to distributed address space of different processes

24

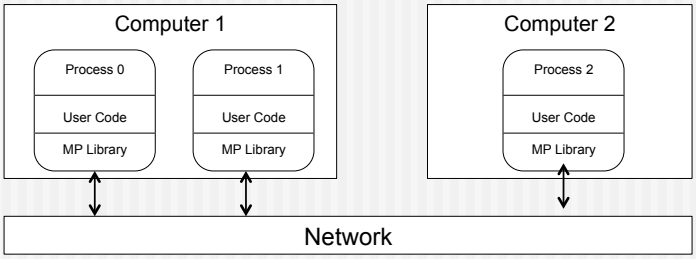
Message Passing

Message Passing Interface (MPI)

25

Message Passing

- Different processes execute in different address space
  - In most cases on different computers
- Inter process communication by exchange of messages over the interconnection network
- Typically facilitated by library calls from within user program



26

MPI Intention

- Originally a wide variety of different message passing libraries available
- MPI provides a specification of a standard library for programming message passing systems
- Interface: practical, portable, efficient, and flexible

27

MPI Goals

- Design of an API (Application Programming Interface)
- Possibilities for efficient communication (Hardware-Specialities, ...)
- Implementations for heterogeneous environments
- Definition of an interface in a traditional way (comparable to other systems)
- Availability of extensions for increased flexibility
- Definition, that is easy to be realized on different kinds of hardware platforms.

28

MPI Forum

Collaboration of 40 Organisations (world-wide):

- IBM T.J. Watson Research Center
- Intels NX/2
- Express
- nCUBE's VERTEX
- p4 - Portable Programs for Parallel Processors
- PARMACS
- Zipcode
- PVM
- Chameleon
- PICL
- ...

29

Development History

- **1992**:
  - "Another" idea for a message passing standard
  - (Pseudo-Standards widely available: PICL, PVM, ...)
  - Initiation of the MPI-Consortium
- **May 1994**:
  - Version 1.0 published
  - Intentionally conservative regarding advanced and controversial features
  - Already over 120 Functions
- **June 1995**: Version 1.1
- **July 1997**: Version 2.0
  - Not many implementations are fully MPI2 compliant!

Need to look closely in the MPI library documentation to see if it's thread safe!

30

MPI Programming Model

- **Parallelization**:
  - Explicit parallel language constructs (for communication)
  - Library with communication functions
  - MIMD (multiple instructions streams over multiple data streams)
  - No (automatic) synchronization of processes
- **Programming Model**:
  - SPMD (single program multiple data)
  - All processes load the same source code
  - Distinction through process number

31

MPI Program

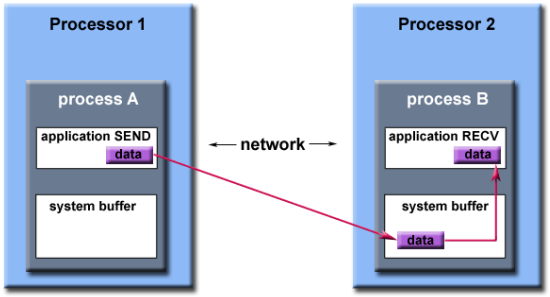
... a sequential program,
that is executed on multiple processes, which
communicate with each other!

32

Functionality

Basically:

- Functions to initialize and run processes on different processors
  ⇨ Process Creation
- Functions for sending and receiving of messages
  ⇨ Send (Message)
  ⇨ Receive (Message)



Path of a message buffered at the receiving process

MPI Functionality

- Process Creation and Execution
- Queries for system environment
- Point-to-point communication (Send/Receive)
- Collective Operations (Broadcast, ...)
- Process groups
- Communication context
- Process topologies
- Profiling Interface

34

Parallel Languages

HPF

PGAS Languages

35

Drawback of Threads and MP

- Threads and message passing are low level programming models
- It's the responsibility of the programmer to parallelize, synchronize, exchange messages
- Rather difficult to use

- Ideally we would like to have a parallelizing compiler that takes a standard sequential program and transforms it automatically into an efficient parallel program
  - In practice static compiler analysis cannot detect enough parallelism due to conservative treatment of dependencies

36

Parallel Languages

- Explicit parallel constructs
  - Parallel loops, array operations, …
  - Fortran 90
- Compiler directives
  - "Hints" to the compiler on how to parallelize a program
  - OpenMP, HPF
- Directives are typically interpreted as comments by sequential compilers
  - Allows to compile parallel program with sequential compiler
  - Eases parallelization of legacy applications

37

High Performance Fortran

38

HPF

- Extension of Fortran 90
- Directive based
- Developed by High Performance Fortran Forum
  - First version 1993
- Data parallel model with main directives for
  - Data alignment and distribution
  - Parallel loops

39

Main Directives

- Data distribution

```
!HPF$ PROCESSORS(2,2)
!HPF$ DISTRIBUTE A(block,block) onto P
!HPF$ ALIGN B(I,J) WITH A(J,I)
```

- Parallel loops
  - F90 array syntax

```
A = B
```

  - forall (made it into Fortran 95 standard)

```
forall (i=1,n)
```

  - Independent

```
!HPF$ INDEPENDENT
      DO i=1,n
```

- Many additional directives and clauses

40

HPF Critisism

- Despite the elegance HPF had in describing parallel programs it never made it into wide-spread use
- For real-word problems HPF compilers had difficulties to generate efficient code
- Later editions of HPF, particularly HPF 2 tried to overcome these limitations with additional constructs, e.g.
  - Extrinsic procedures to escape to message passing
  - Better definition of overlap areas
  - Better support for irregular access patterns
  - But only supported in research compilers
- Several HPF features made it into (and are still being considered for) the Fortran standard

41

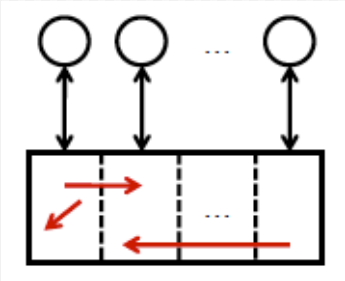
Partitioned Global Address Space Languages (PGAS)

Co-Array Fortran
Unified Parallel C
Titanium
…

42

PGAS

- Tries to overcome the issues shared address space approaches have on distributed memory machines
- Provide a single address space over the physically distributed memory
  - But programmer needs to be aware whether their data is local or remote
- Runtime systems exploit often advanced features like one-sided communication (Cray shmem etc.)



Co-Array Fortran

- Developed in the late 90s (original name was F--)
- Principle idea is language extension to Fortran called *co-array*
- Co-array is the mechanism for interprocessor communication ("co" stands for communication)
- Using square brackets for co-arrays [] indicating how data is distributed over processors

```
X         = Y[PE]   ! get from Y[PE]
Y[PE]     = X       ! put into Y[PE]
Y[:]      = X       ! broadcast X
Y[LIST]   = X       ! broadcast X over subset of PE's in array
                    ! LIST
Z(:)      = Y[:]    ! collect all Y
S = MINVAL(Y[:]) ! min (reduce) all Y
B(1:M)[1:N] = S   ! S scalar, promoted to array of shape
                    ! (1:M,1:N)
```

44

Unified Parallel C

- Developed around 2000
- Array variables can be declared *shared* or *private* where shared variables are distributed using cyclic or block-cyclic distributions
- Parallelism is expressed using forall loops

45

Titanium

- Java extension with a memory model similar to UPC
- Change Java memory management (no garbage collection) and restrict several other Java features to gain performance
- Add foreach loop for expressing parallelism

46

Summary

- Different programming models at different levels
- To compare we need to assess
  - Correctness
  - Performance
  - Scalability
  - Portability

47

Correctness

- Writing correct sequential programs is difficult
  - Adding parallelism doesn't make it any easier
  - Danger of deadlocks, race conditions, etc.
- Threads and Message Passing put the burden of generating correct code on the programmer
  - Locks and send/receive can result in race conditions and dead locks
  - Programming is quite error prone
- Parallel loops (OpenMP, PGAS, HPF) are typically deterministic
  - Much simpler and easier to understand

48

Performance

- Threads have to deal with cross-thread dependencies that impact performance
  - But they look like ordinary memory accesses
  - Locality is not encouraged
- OpenMP provides only simple parallel construct
  - Cannot express all potential parallelism exploitable
- MPI forces to think about locality
  - Message passing adds quite some overhead
  - Coarse grained parallelism
- PGAS languages improve upon MPI but still leave details of local vs. global perspective to the programmer
  - Risk the programmer focuses too much on the local program

49

Portability

- Threads require hardware shared memory support limiting portability
- MPI is in principle portable to almost all machines but many differences in implementations exist thus performance is not portable
  - Programmers tend to tune their MPI program to specific hardware
- PGAS languages improve upon MPI but still have similar issues
  - Runtime system support is key
  - E.g. Hardware support for one-sided communication

50

What will the future bring

- We will have to live with heterogeneity
- Worst case:
  - Many-core CPUs with vector-units and graphics co-processors
  - Combined to SPM nodes
  - Clustered in a DMMP
  - So up to 5 different levels of parallelism
- MPI alone won't make the day!
- New languages have been developed
  - Chapel (Cray), Fortress (SUN), X10 (IBM)
- New GPU languages are being developed
  - CUDA, OpenCL, OpenACC

51

Summary and Outlook

- Overview of Parallel Programming
  - Hardware
  - Theory - key concepts
  - Software
  - Programming Approaches
- Second part of the course will focus more on multicore issues

52

Further Readings

- **Principles of Parallel Programming,** Calvin Lin and Lawrence Snyder
  - Part 3 - Parallel Programming Languages
- **Pthreads Tutorial,** https://computing.llnl.gov/tutorials/pthreads/
- **OpenMP Tutorial,** https://computing.llnl.gov/tutorials/openMP/
- **MPI Tutorial,** https://computing.llnl.gov/tutorials/mpi/

53