COMS/CPRE 425
Spring 2005
sort of
Lecture 31

Ricky A. Kendall
rickyk@cs.iastate.edu
Logistics

- **Presentations start Monday 4/18**
  - Attendance to lecture 4/18 to 4/29 is required!
  - If you miss I better get an email before you miss and why you are missing lecture!
  - It is up to me to decide if it is “excused”
  - Will impact YOUR project Grade!

- **Final Exam**
  - Monday May 2\textsuperscript{nd} 9:45am – 11:45am
  - Expect it to be similar to the Midterm
  - Expect it to be twice as long as the Midterm
  - Closed book, closed notes, no calculators
  - Note the math usually works out remember how to cancel with fractions!
Examination Guidelines

• DO NOT start the exam until told to do so.
• When told to stop students still writing will loose 20% for every 10 seconds they continue to write.
• No calculators or other written/electronic material may be used
  ➢ Written material includes your neighbor’s exam!
• The Exam will be printed in duplex mode (e.g., “double sided”) missing half of the exam because you didn’t turn the page over is no excuse!
  ➢ You have been warned 😊
• Write your name on at least each sheet!!!!
• Write your answers in a legible fashion; if Bin or I cannot read an answer then it is wrong!
  ➢ If you leave a question blank it is impossible to give any partial credit for that question!!!!
  ➢ If you need additional sheets of paper then ask for them!
    ▪ Make sure you write your name and the question number for each response on separate sheets.
• Answer all the questions you know first then work the rest!!
CS425 Course Goals

• What I hope you HAVE LEARNED
  ➢ Proper Programming Practices.
  ➢ Ability to understand and design high performance and parallel algorithms.
    ▪ Basically to develop a tool kit to use in your next “phase” of life.
      ♦ Understand which tool to use!
  ➢ Ability to implement algorithms using a variety of high performance and parallel programming models.
  ➢ Respect for the “Art of Programming.”
    ▪ An important part of Computer Science is the application of the theoretical to the practical.
The Programming Process

- Requirements
- Specification
- Code Design
- Code Development
- Testing
- Debugging
- Release
- Maintenance
- Revision
Matrix Multiply

• Why we are using this algorithm?
  - At the core of many other linear algebra algorithms.
  - Target core algorithm for high performance applications.
  - It is the “hello world” kind of program for high performance computing
  - Simple enough that it is easy to understand.
  - Complex enough that it exposes the nuances of most programming models.
  - Usually will lead to high performance.
    - O(N³) compute with O(N²) communication
  - Can be done in many different ways sequentially
    - Simple loops
    - Vector operations
    - Blocked operations
Matrix Multiply (Blocked)

• Has a more complex loop structure
  ➢ Turns 3 loops into essentially 6 loops
  ➢ This would be a good to understand.
    ▪ You have to do it for the Homework
    ▪ You might be requested to do this in the future (e.g., on an exam).
Blocking a loop

• **Loop to block:**
  ```
  for (j=0; j < N; j++)
  {
      x = f(j);
  }
  ```

• **To block the loop you have to determine**
  - The size of the block to use
  - The mechanism to implement the appropriate values in the code.
  - This makes one loop into two loops.
  - The size of the block must be small enough to minimize cache misses
  - The size of the block must be large enough to get good performance!
Blocking a single loop

• Loop to block: (j goes 0,1,2,3,4,..., N)
  for (j=0; j < N; j++)
  { x = f(j); }  

• This becomes (one of many ways):
  block_size = 4; num_blocks=N/block_size+1;
  for (jb = 0; jb<num_blocks; jb++) {
    jlo = jb*block_size;
    jhi = MYMIN(jlo+block_size , N)
    for (j=jlo;j<jhi;j++) { x = f(j);} }  
  (j goes 0,1,2,3,  4,5,6,7,  8,9,10,11, ... N)
The Scope of HPC

• What is a High Performance Computer or a supercomputer?
  ➢ There is no static definition.
  ➢ It implies bigger, better, faster, and more expensive.

• Basically covers from the palmtop to the teraflop systems available.
  ➢ Most based on RISC processors
  ➢ COTS or NOW clusters
    ▪ Commodity of the Shelf
    ▪ Network of Workstations
  ➢ Vector Supercomputers
  ➢ SMP systems, MPP systems
  ➢ IBM Blue Gene System
  ➢ Japan’s Earth Simulator
Supercomputers

• Integrate many resources into a single system
• Employ at least some bleeding edge technology
  ➢ First of a kind
  ➢ One of a few
• Have the highest performance or
• Offers the promise of the highest performance
  ➢ What determines the reality?

Applications and utilization
Similarities of Parallel and Distributed Computing

• Multiple processors are involved
• Multiple processes that cooperate via some mechanism are operating concurrently.
• The processors (and processes) are interconnected by some network fabric (network protocol).
  ➢ Fast Ethernet (MPI, MPICH, … etc.)
  ➢ Myrinet (GM)
  ➢ Quadrix (SC Software)
  ➢ SCI (Scali Software).
• Parallel computing is often considered a sub-discipline of distributed computing.
  ➢ One of the more difficult ones 😊
Flynn’s Taxonomy

- **SISD** (von Neumann)
- **SIMD** (MasPar)
- **MISD**
- **MIMD** (IBM SP)

**DATA**

**INSTRUCTION**

- Single
- Multiple
Parallelizing your application is considered RE-BUGGING your code!!!
Programming Parallel Computers

• **Essentially 4 ways to attack the issue**
  1. Extend an existing compiler to translate sequential programs into parallel programs
  2. Extend an existing language with new operations to express parallelism
  3. Add a new parallel language layer on top of an existing sequential language.
  4. Define a new language and compiler system.
How do you write applications for these MIMD-DM systems?

- **SPMD**
  - Single Program Multiple Data
    - Flynn-ism
  - Do something different based on Process or Thread ID.

- The same program executes on all processors.
  - Via processes, threads, or both.

- The method and speed of addressing memory on each node is the key to performance.
Memory Hierarchy on a Node

- Registers
- Cache(s)
- Main Memory
- Virtual Memory

The diagram illustrates the relationship between speed and cost, with size as a parameter. As size increases, speed and cost decrease.
Unifying Theme

• The only difference in parallel computer systems is the access to the memory hierarchy of the system.

• Non-Uniform Memory Access
  - Applies to both Shared Memory and Distributed Memory MIMD parallel supercomputers.
  - Not only speed (Bandwidth & Latency) but the method of access as well!
MPP Memory Hierarchy

- Registers
- Cache(s)
- Main Memory
- Virtual Memory
- Remote Main Memory
- Remote Virtual Memory

Speed vs. Size
Programming Models

• **Shared Memory**
  - process based
  - thread based (OpenMP, Pthreads)

• **Message Passing**
  - MPI, MPI-2, PVM etc.

• **Combined Message Passing and SMP**
  - MPI and OpenMP

• **Globally Addressable (NUMA)**
  - Global Arrays
  - Unified Parallel C
  - Co-Array Fortran
Accessing Memory

- **Shared Memory:**
  \[
  A[i] = B[j] + C[k];
  \]

- **Message Passing**
  \[
  \text{Send}(B); \ \text{Receive}(B); \ \text{Send}(C); \ \text{Receive}(C);
  \]
  \[
  A[i] = B[j] + C[k];
  \]

- **Combined Message Passing and Shared Memory**
  \[
  \text{BUT you must know where the data is!}
  \]

- **Globally Addressable (NUMA)**
  \[
  \text{get}(B); \ \text{get}(C);
  \]
  \[
  A[i] = B[j] + C[k];
  \]
Scalable Architectures

• A computational resource is scalable if it can expand to accommodate additional demands.

  ➢ Better performance and/or more functionality.
    ▪ A linear increase in power with an increase in resources.
  ➢ Both hardware and software
  ➢ Reasonable cost scaling
    ▪ A factor of N in scaling should require a linear or NlogN cost scaling
    ▪ Can also scale down with a reduced cost
Scalable Architectures (2)

• A resource is scalable …

  ➢ Compatibility
  ➢ The same software, hardware, applications should work as the system grows.
  ➢ Best done scaling all resources but should work just adding a single resource
    ♦ A processor upgrade
    ♦ A memory upgrade
    ♦ Which is the more costly of these two?
Architecture Nomenclature

• **Macro-architecture**
  - The overall structure of the computational components
    - The “multiple-node” or repeated unit architecture.
    - The interconnect for the nodes
    - Multiple system images (single image in special cases).

• **Micro-architecture**
  - The structure of a single instance of a computational component.
    - The node itself.
    - SMP or processor
    - Always a single system image
Dimensions of Scalability

- **Resources**
- **Applications**
- **Technologies**
Generation or Time Scalability

- Five sub-systems of any computer:
  - **CPU**
    - Doubles in speed every 18 months (Moore’s Law)
  - **Memory**
    - Doubles in size/density every 3 years
  - **Communication (bus, switch, network)**
    - Doubles in bandwidth and reduced latency every 3 to 10 years.
  - **Secondary Storage (disks)**
    - Doubles in size every 2 to 3 years
    - Doubles in bandwidth every 5 to 10 years.
  - **Software (OS, Compilers, Languages)**
    - Improves over 1 or two generations of other components.
    - Sometimes rewritten for each generation.
Application Tradeoffs

• Algorithms can be tuned by making tradeoffs between computer sub-systems
  ➢ The block size in the Example Homework Problem is a “tunable” parameter.
  ➢ Trade reduced computation for storage
  ➢ Trade reduce storage for computation
  ➢ Trade reduced communication for storage.
  ➢ Trade increased computation for better communication performance
Parallel Computer Models

- Previously we saw Flynn’s Taxonomy (a Model)
- The von Neumann SISD (a Model)
- An abstract view of the architecture or programming mechanism
- Characterizes capabilities of the machine
  - Usually with some simplifications
    - Ignores actual processor implementation(s)
    - Ignores communication network topology
- Precise enough to think about performance but ignores explicit implementation details.
- Helps users/developers understand the performance of their applications.
Parallel Computer Models

• Defined by coupled, semantic and performance attributes.
  ➢ Think of a parallel computer model as a way of grouping processes to concurrently solve a problem.

• Semantic Attributes
  ➢ Needed to effectively write a parallel program
  ➢ “thinking in parallel”

  ➢ Specific Attributes:
    ▪ Homogeneity
    ▪ Interaction Mechanism
    ▪ Memory Model
    ▪ Synchrony
    ▪ Address Space
Performance Attributes

• Definitions
  - Machine Size
  - Sequential Time
  - Parallel Time
  - Speed
  - Peak Speed
  - Speed Up
  - Efficiency
  - Startup time
  - Bandwidth
  - Asymptotic Bandwidth

Definitions:
- Machine Size
  - \( n, N_p, N_{proc} \)
- Sequential Time
  - \( T_1, T_s \)
- Parallel Time
  - \( T_n, T_p \)
- Speed
  - \( P_n, R_n, MFR = \text{flops/time} \)
- Peak Speed
  - \( P_{\text{peak}}, R_{\text{macho}} \)
- Speed Up
  - \( S_n = T_s/T_p = T_1/T_n \)
- Efficiency
  - \( E_n = S_n/n = S_n/N_p \)
- Startup time
  - \( t_0, \tau_0 \)
- Bandwidth
  - \( r \)
- Asymptotic Bandwidth
  - \( r_\infty \)
Kinds of Operations

• **Computation**
  - Anything that exists in a traditional sequential program
  - Arithmetic/logic
  - Control Flow (branches)
  - Data transfer (local memory copy, disk I/O)

• **Parallelism**
  - Startup/shutdown of processes
  - Grouping or work flow designation (What do I do?)

• **Interaction**
  - Data transfer among processes
  - Synchronization
Operations Context

• Operations can be either:
  ➢ Explicit
    ▪ Appears as a construct in the program
    ▪ Handled primarily by the application developer
    ▪ E.g., exit(), A = B+C, fork()
  ➢ Implicit
    ▪ Does not appear as a construct in your program
    ▪ May appear as a construct in libraries you use:
      ♦ MPI, GA, etc.
    ▪ Handled primarily by the Operating System or your Library Developer
    ▪ E.g., Broadcast implies a synchronization.
Overhead (The pain in Parallel)

• Parallelism
  ➢ Process initiation/termination
  ➢ Constructs from Libraries you may use

• Communication
  ➢ Data exchange operations
    ▪ Collective
    ▪ Non-collective (e.g., point-to-point)

• Synchronization
  ➢ Synch operations

• Load Imbalance
  ➢ Some processes not doing useful work
Abstract Machine Models

- A more global view of both semantic and performance attributes
- Three models to examine, increasingly more complex.
  - Parallel Random Access Machine Model (PRAM)
  - Bulk Synchronous Parallel Model (BSP)
  - Phase Parallel Model (PPM)
Computational Complexity

• Assuming a PRAM model (no overheads) and three algorithms that have the following computational complexities:
  - \( A: 7N_p \quad O(N_p) \)
  - \( B: (N_p \log N_p)/4 \quad O(N_p \log N_p) \)
  - \( C: N_p \log \log N_p \quad O(N_p \log \log N_p) \)

• Consider both the real and traditional BigO scaling.
  - Note BigO implies asymptotic behavior \( (N_p \to \infty) \)

• \( N_p \log N_p \) is “good” scaling
Scalable Design Principles

• Based on lessons learned from successful and failed
  ➢ Parallel computer designs
  ➢ Parallel application designs

• Both computers and applications are complex engineering processes

• Principle of independence

• Principle of balanced design

• Principle of design for scalability

• Principle of latency hiding.
Why is Parallel Programming Hard?

- More complex than sequential programming
  - Is sequential plus explicit concurrency
- More than one basic programming model
  - Concurrency hidden; the compiler does not handle sequential programs
  - Usually explicit in a parallel programming model
- Software tools are less mature or non-existent.
- Languages are adapted/modified from sequential programming models.
  - FORTRAN and C are the primary languages in both paradigms.
- Lack of practical experience.
  - Need to train the current/next generation programmers
Shared Memory Programming

• The most simple parallel mechanism available.
  - All nodes/processes see all of the memory 😊
  - Many algorithms can be easily implemented.
  - Either a process or thread based mechanism.
  - There are three main methods in use today
    - System specific shared memory mechanisms
      ♦ Cray, Sun, SGI, Fujitsu, NEC, Hitachi
    - Pthreads (POSIX) thus “portable”
      ♦ Runs on most systems
      ♦ C only
    - OpenMP
      ♦ New standard designed for SMPs.
      ♦ Works with C/C++ and FORTRAN
Multithread Programming

- Multithreading (MT) is a technique that allows one program to do multiple tasks concurrently.
  - A technique that has been around since the 1970s.
  - Languages such as Ada implement tasks as multiple threads.
  - The MT programming paradigm became accepted and standardized in the 1990s.
    - In 1991 only a few OSs had a user level threads library. (Solaris)
    - In 1997 most had a threads library available to users.
  - The SMP technology was really a boon to the threads programming paradigm.
    - Combined concurrency with parallelism.
- There is a lot of interaction between MT and OSs.
Value of Threads

- Performance gains from SMP hardware
- Increased Application Throughput
- Increased Application Responsiveness
- Eliminates IPC
- Efficient use of resources
- Ability to make use of concurrency
  - Leads to a more modular program paradigm
- Same executable/OS works on
  - Uniprocessors
  - SMPs
- Single source code for multiple systems/platforms
Load Balancing

• The process of dividing work done by each available processor into approximately equal amounts.

• In a multiuser environment, the number of available processors is constantly changing.

• Dynamic load balancing done by creating small granularity parallelism.
  - For example, work is allocated to each task one iteration at a time.
    - This is not always the best 😊
  - A loose example of this is the Unix Time Sharing system.
Load Balancing [2]

• The relationship between load balancing and the extent of parallelism is as follows:
  - The higher the extent of parallelism, the easier it is to balance the workload evenly across the processors.
  - Small granularity parallelism is easier to balance across available processors than large granularity parallelism.
  - Small granularity parallelism generates more overhead than large granularity parallelism.
  - Synchronization is required each time a chunk of work is allocated to a processor.
  - You must evaluate the trade-offs between load balancing and overhead.
Load Balancing [3]

- You can resolve a load imbalance problem by
  - adjusting the size of work for each task,
  - reorganizing a nested loop structure,
  - selecting an intermediate level loop to be parallelized instead of the outer loop.
  - Make sure there is enough work to parallelize
OpenMP

- **Industry Standard developed in 1997**
  - This is relatively new
  - Few free implementations for C, none for Fortran
    - Most based on Pthreads or Java Threads.
  - Based on common thread based models in existence
    - Solaris, SGI, Cray parallel thread models.
    - Pthreads
    - More structure imposed than a general thread model.
  - Goal to be the standard shared memory programming model for single address space computing.

- **OpenMP Architecture Review Board (ARB)**
  - Runs the show
  - http://www.openmp.org
OpenMP Advantages

• Ability to parallelize small parts of an application at a time.
  ➢ Start with the most time critical ones 😊
• Simple or complex algorithms can be expressed using OpenMP
  ➢ Proportional amounts of work for each
• The size of the code is only marginally increased
  ➢ 5 to 20 percent
• Reasonably clear expression of parallelism so code is “easy” to read.
  ➢ Not many extra calls.
• Integrated debuggers starting to become available.
• Single source code for both sequential and parallel libraries
  ➢ Non OpenMP compilers ignore the OMP directives
OpenMP Programming Model

• The Application Programmer Interface or API is a combination of:
  ➢ Directives
  ➢ Runtime library routines
  ➢ Environment variables
Communication/Data Environment

• Threads have the shared memory of the process
  ➢ Separate stacks and program counters
• Variables in a program have different scope or execution context.
  ➢ Shared
    ▪ Among all threads
  ➢ Private
    ▪ Each thread has its own copy
  ➢ Reduction
    ▪ Both a private and shared context.
Scheduling of Loops

- **Static**
  - Choice of which thread performs a particular iteration is solely a function of the iteration number and the number of threads
    - Statically assigned at the beginning of the loop
    - Load imbalance may be an issue if each task has varying work.
    - Lowest or low overhead.

- **Dynamic**
  - The assignment of threads varies at runtime and with the implementation.
    - Each thread gets another set of iterations (a chunk of work) after it completes the current set of work.
    - Load balance of work possible.
    - Higher overhead
Scheduling the loops.

- The default is “static”
- Static can have a chunk size.
- Dynamic with a chunk size
- Guided special dynamic
  - With a minimum chunk size
- Runtime
  - Any of the above
  - MUST have environment variable set
    - `% setenv OMP_SCHEDULE "static,10"`
- #pragma omp parallel for schedule(type[,chunk])
- !$omp parallel do schedule(type[,chunk])
The problem with OpenMP

• **Removing Data Dependencies.**
  - You have a data dependence when
    - One statement reads or writes a memory location
    - A second statement reads or writes the same location
    - And at least one of the statements writes the location!

• **Example:**
  - for(j=1;j<n;j++)
  -  \[a[j] = a[j] + a[j-1];\]

• A data race because in parallel two or more threads can read or update the same location!
How do you detect data dependencies?

• **Examine memory locations**
  - Is the variable only read and never assigned within the loop body (or parallel construct)
    - If so, there are no data dependencies
  - For each location.
    - Is there exactly one iteration that accesses the location?
      ♦ If so there is not dependence
      ♦ Otherwise there may be a dependence!

• **Loops are parallelizable IFF**
  - All assignments are to arrays
  - Each element is assigned by at MOST one iteration.
  - No iteration reads elements assigned by any other iteration.
Second Order Data Race

- **Actual data dependencies are a problem.**
  - Near data races are a problem too.
- **This is called False Sharing**
  - Occurs when 2 or more threads modify different data elements which happen to be in the same cache line!!
    - Each processor has its own cache.
  - Cache coherence is a hardware aware operation on SMPs.
    - Thread one modifies `local_count[0]`
      - Invalidates all copies of `local_count` cache line
    - Thread two modifies `local_count[1]`
      - Invalidates all copies of `local_count` cache line.
  - This will work but it thrashes cache and thus degrades performance.
Data Dependencies

• **Analysis**
  - Must include the lexical extent of the loop or parallel construct
  - Also must include the entire dynamic extent
    - Analysis of any function called is required too!
    - Things that look private may really be shared
      - As per examples of the default scoping of variables
Dependence Classification

• **Loop-carried dependence**
  - Mostly what we have looked at!
  - Whether or not the two statements involved in the dependence occur in different iterations of a parallel loop!!

• **Non-Loop-Carried dependences**
  - Do not cause a data race
  - Statements are executed in order.
    - In a single iteration of a loop!
    - Just like normal code or serial code in the master thread.
Dataflow dependencies

- **Do two statements communicate values through the memory location**
  - First statement is S1
  - Second statement is S2

- **Flow Dependency**
  - S1 writes the memory location
  - S2 reads the location
  - The value read by S2 serially is the same as written by S1!
  - The data S1 writes “flows” to S2.
Removing the Dependencies

• **Must remove all loop-carried dependencies to parallelize the loop**
  - Change the scope of the variable
  - Transforming the loop source code
  - Doing both.
  - May not be possible in all cases!!!
    - Might have to recode or switch algorithms 😞
  - Must be careful to:
    - Not change the semantics of the loops
    - Not introduce additional loop-carried dependencies
      - If you do you need to remove them too 😊
The Overview

- **Amdahl’s Law**
  - Decide whether a program merits a parallel implementation.
  - Speedup

- **Gustafson-Barsis’s Law**
  - Performance of a parallel program.
  - Scaled speedup

- **Karp-Flatt metric**
  - What is the principle barrier to performance
    - Sequential code?
    - Parallel overhead?
  - Experimentally determined speedup
Amdahl’s Law

\[ f = \frac{\sigma(n)}{\sigma(n) + \varphi(n)} \]

\[ \psi(n, p) \leq \frac{1}{f + \frac{(1 - f)}{p}} \]
Amdahl Effect

- For a fixed number of processors speedup usually increases with the problem size.
  - Not getting good speedup increase the problem size and you will see better speedup!
- The overhead $\kappa(n,p)$ usually has a lower problem size and processor count dependence than the parallel algorithm component, $\varphi(n)$
- For example:
  - $\kappa(n,p) = O(500*\log n + (n/25)*\log p)$
  - $\varphi(n) = O(n^2)$
Gustafson-Barsis’s Law

• Treats time as the constant and motivation is the scaled problem size.
  - How big of a problem can I solve in a fixed amount of time.
  - Ignores $\kappa(n,p)$ the overhead

• Opposite of Amdahl’s Law.

• Assumes a fraction of time, $s$, spent in parallel code doing sequential operations

$$\psi (n, p) \leq p + (1 - p)s$$

• Often called Scaled Speedup.

• John Gustafson, ISU grad, SCL Scientist.
Karp-Flatt Metric

- The experimentally determined serial fraction is defined as the ratio of the serial time plus the overhead, $\kappa(n,p)$, to the time to compute on one process with the parallel code.

$$ e = \frac{[\sigma(n) + \kappa(n,p)]}{T(n,1)} $$
Isoefficiency

• **Efficiency is a function of the problem size and the number of processors**
  
  ➢ For a fixed sized problem
    ▪ Efficiency decreases as the number of processors increase
    ▪ Efficiency increases as the problem size increases
  
  ➢ The goal is to figure out how to scale the problem size and the number of processors with a constant efficiency!
Big Picture Comparison

- **Single thread or serial code**
  - Simple interactions between code and data
    - Compilers are our friends 😊

- **OpenMP shared memory programming**
  - Data is shared among all threads
  - Implicit thread creation and work sharing

- **Pthreads shared memory programming**
  - Data is shared among all threads
  - Explicit thread creation and work sharing
    - You the programmer schedule “EVERYTHING”

- **Message Passing Programming**
  - Data is not shared among processes
    - Data movement is explicitly scheduled among processes
  - Explicit process creation and work sharing
    - You the programmer schedule “EVERYTHING”
State transitions

- **Ready**
- **Running**
- **Blocked**
- **Terminated**

**Transitions:**
- Ready → Running
- Running → Scheduled
- Scheduled → Ready
- Running → Preempted
- Running → Canceled
- Running → Finished
- Finished → Terminated
- Blocked → Resource Wait
- Resource Wait → Wait Over
- Wait Over → Ready
- Preempted → Blocked

**Notes:**
- 4/8/2005
- ComS 425
- Spring 2005
- 64 of 158: Lecture 31
Kernel Scheduling Models

- Many Threads to one LWP (N:1)
  - Many user space threads
  - All execute on a single LWP
  - Little speed up 😊
  - Blocking system calls (all-stall)

- One Thread per LWP (1:1)
  - Use multiple CPUs
  - Non-blocking systems calls possible.
  - Thread creation requires LWP creation.

- Many Threads on Many LWPs (M:N)
  - Threads multiplexed onto equal or smaller number of LWPs.
  - Some systems have the ability to request (1:1) thread spaces
Terminology

• What are Thread Safe Libraries?
  ➢ All shared resources by the library routines are protected from concurrent access by locks

• What are reentrant functions?
  ➢ No static data is held over successive calls.
  ➢ No pointers to static data are returned.
  ➢ Push, push, push, push, push, just too pushy
Pthreads programming model

- Explicit thread creation and termination!
- Explicit thread code!
- SPMD
  - Still a single program that executes everywhere
- Shared memory programming model
- ONLY available and portable in the C/C++ language
  - The standard POSIX interface.
  - A FORTRAN standard exists but only implemented by IBM
- Explicit shared and private programming access.
  - Pointer management from file scope and how
    stack/automatic variables are constructed
  - The programmer must know the context!!
Thread Attributes: scope

- Defines the scheduling contention scope for the created thread.
- **PTHREAD_SCOPE_SYSTEM**
  - means that the threads contend for CPU time with all processes running on the machine.
- **Bound Threads**
- **PTHREAD_SCOPE_PROCESS**
  - means that scheduling contention occurs only between the threads of the running process
  - thread priorities are interpreted relative to the priorities of the other threads of the process, regardless of the priorities of other processes
- **Unbound Threads**
- **Default values:**
  - Linux PTHREAD_SCOPE_SYSTEM.
  - Solaris PTHREAD_SCOPE_PROCESS.
Mutexes

- Has two states locked or unlocked
- Has two functional operations lock and unlock
- Only one thread can hold the lock
  - Any other thread trying to lock it will wait until it is unlocked.
    - Put into a “blocked” state
  - The lock holding thread has exclusive right to the critical section of code that is controlled by the lock.
    - This can be doing operations
    - Accessing data (reads)
    - Initializing setting data (writes).
- Can be either statically or dynamically allocated.
Lock Contention.

- When multiple threads are all “banging” on the same lock.
- One solution is to allocate an array of local_counts for use by each thread.
  - As I did in the pi example I handed out.
- This will reduce the lock contention but will not perform optimally.
  - Why?
False Sharing

- Occurs when 2 or more threads modify different data elements which happen to be in the same cache line!!
  - Each processor has its own cache.
    - Cache coherence is usually a hardware aware operation on SMPs.
  - Thread one modifies local_count[0]
    - Invalidates all copies of local_count cache line.
  - Thread two modifies local_count[1]
    - Invalidates all copies of local_count cache line.
  - This will work but it thrashes cache and thus degrades performance.
Pthreads

• There is another problem with mutexes
  ➢ Deadlock where a lock will never be obtained because of the code design.
  ➢ A 1-800-OOPS-BUG kind of thing.
  ➢ Deadlock is also possible in message passing so keep this in your brain even when we are done with pthreads 😊
Deadlock with Mutexes

Thread 1

Pthread_mutex_lock(l1);
Pthread_mutex_lock(l2);

Thread 2

Pthread_mutex_lock(l2);
Pthread_mutex_lock(l1);

This is a \textit{bad} thing; careful planning is needed for lock hierarchies.
Condition Variables

- **Are always protected with a mutex.**
  - Why?
  - Race Conditions 🙃
  - One thread might be testing the condition while another is updating the condition!!!

- **There are implicit and explicit**
  - Locks of mutexes
  - Unlocks of mutexes

- **Explicit**
  - Those you code

- **Implicit**
  - Those the Pthreads library does for you 😊
A diagram of condition variable use

Thread waiting on condition

Thread that sets condition

LOCK

Condition ?

UNLOCK

(lock)

SLEEP

Wakeup/Signal

Continue

Continue

LOCK

Update Condition

UNLOCK

4/8/2005

ComS 425

Spring 2005

75 of 158: Lecture 31
Limiting the number of threads

• The goal is to provide a user defined maximum number of threads to manage resource utilization.

➢ Even with OpenMP we have to implicitly be concerned with this.
  ▪ On the NERSC systems the 1500,1501,1499 case was unable to run with the default data and stack sizes!
    ♦ Remember -bmaxstack and -bmaxdata?

➢ You have to balance the granularity manually.
  ▪ Essentially the “work” load per task

➢ By limiting the number of threads you also minimize the parallel overhead.
Governing Thread Count

• There are many ways to govern the number of threads.

  ➢ Generate a pool of threads
    ▪ All working threads are created as a part of the pool and wait for work to be assigned.
      ♦ Essentially a queue data structure
    ▪ Work is dynamically assigned to them.
    ▪ Once the work is completed then the thread returns to the pool and is available for further tasks.
    ▪ Fully dynamic mechanism.

  ➢ Generate a fixed number of threads and loop work assignments to them.
    ▪ Instead of a queue this is done using a stack based mechanism.
    ▪ Generate the threads with work assigned.
    ▪ Wait for the “stacked” thread to complete
    ▪ Assign the NEXT task to the thread.
    ▪ Not a fully dynamic mechanism.
Stack Example.
How the pool works

init

Add Work

Collect results

Thread Thread Thread Thread Thread Thread Thread Thread Thread

destroy
Where to parallelize? [1]

- Loop over i blocks
- Loop over j blocks
- Loop over k blocks

Thread Work
Where to parallelize? [2]

- Loop over i blocks
- Loop over j blocks
- Loop over k blocks

$$C_{i,j} += A_{i,k} \times B_{k,j}$$

Thread Work
Where to parallelize? [3]

- Loop over \( i \) blocks
- Loop over \( j \) blocks
- Loop over \( k \) blocks

\[
C_{i,j} = A_{i,k} \times B_{k,j}
\]
Message Passing History

• Loosely based on Message Queues of System V Interprocess Communication Protocols (IPC).
• Common practice on early parallel systems.
  ➢ Usually Hypercube, ring, or Mesh Networks.
  ➢ Home grown message passing libraries.
  ➢ Vendor Specific libraries.
• Intel Hypercube systems
  ➢ Some of the first general purpose systems available to scientific community
  ➢ Did much to further development of Message Passing programming systems.
General Message Passing Model

- **Multiple instances of execution (SPMD).**
  - Separate processes with their own thread of control.
  - May execute different code
  - Both control and data parallelism are supported.

- **Asynchronous Parallelism**
  - Special constructs used to synchronize processes
    - Barriers and collective communication mechanisms

- **Separate Address Space**
  - One process cannot “see” memory of another process.

- **Explicit Interactions**
  - Programmer must code all interactions

- **Explicit Allocation**
  - Workload and data are explicitly assigned to specific processes.
Communication Modes

- Important issues understanding of context
  - How many processes (and possibly threads) are involved.
  - How the processes are synchronized
  - How are communication buffers are managed

- Three modes of operation
  - Synchronous
  - Blocking
  - Non-blocking
Synchronous Mode

send

receive

continue
Blocking Mode

send

continue

buffer

receive
Blocking Mode [2]
Non-Blocking Mode

send

continue

buffer

receive
Non-Blocking Mode [2]

send

buffer

continue

? = receive

buffer

continue

?
**MPI Basic Concepts**

- Processes have separate address spaces
- Communication is copying a part of one process’s address space to one or more other process’s address space.

![Diagram](http://example.com/diagram.png)

- The first process that “has” the data being copied issues the “send” library call the other process issues the “receive” library call.
MPI Communication Modes

• **Standard**
  - The blocking, and non-blocking mechanisms we have discussed operate in the way we described.

• **Synchronous**
  - Fully synchronous mechanism.
    ▪ Helpful for debugging

• **Ready (for sends)**
  - System notified of the receive post
    ▪ Allows the system to use a faster protocol if available.

• **Buffered**
  - Allows a user-controllabble buffering for send operations.
Message Passing support for Libraries

• The communicator context and groups allow the independent development of “other” libraries.

• Libraries can create groups and base operations on these groups

• Tag matching depends upon the communicator context

  ➢ A big problem before MPI

• Independent error handlers etc.
Minimal MPI

- The basic commands that can be used to write a parallel application (e.g., those used by this course – nine routines!)
  - `MPI_Init` Initialize MPI
  - `MPI_Comm_size` Number of processes
  - `MPI_Comm_rank` Which process I am
  - `MPI_Send` Send a message
  - `MPI_Recv` Receive a message
  - `MPI_Bcast` Broadcast data with all
  - `MPI_Reduce` Global reductions
  - `MPI_Finalize` Terminate MPI
  - `MPI_Abort` Handle errors “cleanly”
Basic MPI Timing Routines

• **MPI_Wtime**
  - Returns an elapsed time on the calling processor
  - **Synopsis**
    - `#include "mpi.h"
    - `double MPI_Wtime()`
  - **Return value**
    - Time in seconds since an arbitrary time in the past.

• **MPI_Wtick**
  - returns the resolution of MPI_WTIME in seconds. For example, if the hardware clock is incremented every millisecond, the value returned should be $10^{-3}$
  - **Synopsis**
    - `#include "mpi.h"
    - `double MPI_Wtick(void)`
  - **Return value**
    - Time between clock ticks
How does a broadcast work?
How does a Reduction work?
How does a Allreduce work?
## Scatter Process

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Y[0] to Process 0
- Y[1] to Process 1

## Gather Process

<table>
<thead>
<tr>
<th>Process</th>
<th>X[0]</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Process 0</td>
<td>X[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 1</td>
<td>X[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 2</td>
<td>X[0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process 3</td>
<td>X[0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pseudo-Code for Fox’s Method

• \( i = \text{my process row}; \quad j = \text{my process column}; \)
• \( \text{Nsqr} = \sqrt{\text{Nproc}}; \)
• \( \text{dest} = ((i-1) \% \text{Nsqr},j); \)
• \( \text{source} = ((i+1) \% \text{Nsqr},j); \)
• \( \text{for}(\text{stage} = 0; \; \text{stage} < \text{Nsqr} \; ; \; \text{stage}++ \) \} { 
  ➢ \( k' = (i + \text{stage}) \% \text{Nsqr} \)
  ➢ \text{Broadcast sub-matrix } A_{i,k'} \text{ to process row members}
  ➢ \( C_{i,j} += A_{i,k'} \ast B_{k',j} \)
  ➢ \text{Send } B_{k',j} \text{ to dest}
  ➢ \text{Receive } B_{(k'+1),j} \text{ from source}
• }
Row and column Communicators

• MPI offers three ways to do this.
  ➢ MPI_Group_incl, MPI_Comm_create
  ➢ MPI_Comm_split
  ➢ Topologies graphs and grids
MPI_Comm_split for Fox Algorithm

• Remember that $q = \sqrt{\text{Nproc}}$

• For the row communicators;
  - $\text{Myrow} = \text{my\_rank}/q$;
  - $\text{MPI\_Comm\_split(MPI\_COMM\_WORLD,Myrow, my\_rank,row\_comm);}$

• For the column communicators
  - $\text{Mycol} = \text{my\_rank} \mod q$;
  - $\text{MPI\_Comm\_split(MPI\_COMM\_WORLD,Mycol, my\_rank,col\_comm);}$
Virtual Topologies

- For Fox’s algorithm assume a square grid of processes that are in MPI_COMM_WORLD.
- We need to identify:
  - The number of dimensions (2)
  - The size of each dimension
    - The number of rows and number of columns
      - Square root of number of processes and both are “q”
  - Periodicity in each dimension
    - Is the first element in this dimension adjacent to the last element.
      - For columns YES
      - For Rows not important (broadcast).
  - Do we allow MPI to reorder processes.
Pseudo-code for Homework 5

• Set up process grid, row, and column communicators.
• Allocate patch components
  ➢ A, A',
  ➢ B, B',
  ➢ Ccomp, Canal
• In Gen_A, Gen_B, Gen_C
  ➢ Identify process grid points (i,j)
  ➢ Compute patch associated with grid points
• Broadcast, Multiply, Roll (do Fox algorithm)
• Compare locally Ccomp and Canal
  ➢ Merge results with global operations
• Time wall time of every step.
  ➢ Merge results , min, max, average
Cannon’s Algorithm

<table>
<thead>
<tr>
<th>Process Mappings</th>
<th>Process Column 0</th>
<th>Process Column 1</th>
<th>Process Column 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Row 0</td>
<td>0 (0,0)</td>
<td>1 (0,1)</td>
<td>2 (0,2)</td>
</tr>
<tr>
<td>Process Row 1</td>
<td>3 (1,0)</td>
<td>4 (1,1)</td>
<td>5 (1,2)</td>
</tr>
<tr>
<td>Process Row 2</td>
<td>6 (2,0)</td>
<td>7 (2,1)</td>
<td>8 (2,2)</td>
</tr>
</tbody>
</table>

Cannon’s Algorithm Elemental

1. Process \( P_{i,j} \) initially has \( a_{i,j} \) and \( b_{i,j} \)

2. Align the elements of A and B
   1. Use periodic shifts (e.g., wrap around the edges)
   2. The complete \( i^{th} \) row of A is shifted left “\( i \)” times
   3. The complete \( j^{th} \) column of B is shifted upward “\( j \)” times.
   4. This means that \( P_{i,j} \) has \( a_{i,j+i} \) and \( b_{i+j,j} \) elements

3. On each process \( P_{i,j} \) \( c_{i,j} = \text{local}_a \ast \text{local}_b \)

4. Setup for \( n-1 \) stages (sound familiar)
   1. Shift \( \text{local}_a \) left, \( \text{local}_b \) up
   2. Then \( c_{i,j} += \text{local}_a \ast \text{local}_b \)
Comparison of Results

• The computed C values have the same data layout of the analytical ones.
  - Computing the norm is the same as for Fox’s algorithm.
    ▪ Compute a local difference vector.
    ▪ Compute the local norm of the difference vector
    ▪ Do a sum reduction across all processes to compute the full norm.

• Note that for completeness you must un-align the elements of A and B to get them back to their original position.
  - Or is that de-align???
Cannon’s Algorithm

• The element by element analysis is the same if \( a_{i,j} \) and \( b_{i,j} \) are patches of matrices as well.
  ➢ Just like the Fox analysis!
• No extra storage really needed.
  ➢ Appropriate shifting before, during and after.
  ➢ Can be generalized to non-square matrices
    ▪ Rectangular.
  ➢ For square matrices
    ▪ Grid is the same as fox’s algorithm with respect to coordinates etc.
    ▪ No broadcasts.
    ▪ Only systolic shifts.
  ➢ Has different communication costs than Fox’s algorithm.
Code Fragment for Cannon First Shift

myrow = coords[0]; mycol = coords[1];
A_dest = (mycol - myrow + q)%q;
A_source = (mycol + myrow + q)%q;
B_dest = (myrow - mycol + q)%q;
B_source = (myrow + mycol + q)%q;
if (A_source != A_dest) {
    MPI_Sendrecv(&myA,1,MPI_DOUBLE,A_dest,565,
                 &tA,1,MPI_DOUBLE,A_source,565,MPI_row,
                 &mystatus));
}
if (B_source != B_dest) {
    MPI_Sendrecv(&myB,1,MPI_DOUBLE,B_dest,123,
                 &tB,1,MPI_DOUBLE,B_source,123,MPI_col,
                 &mystatus));
}
Cannon’s Code fragment for stages

```c
for(stage=1; stage<q; stage++ ){
    A_dest   = (mycol - 1 + 2*q) % q;
    A_source = (mycol + 1 + 2*q) % q;
    MPI_Sendrecv(&tA,1,MPI_DOUBLE,A_dest,565,
                  &ttA,1,MPI_DOUBLE,A_source,565,MPI_row,
                  &mystatus));
    tA = ttA;
    B_dest   = (myrow - 1 + 2*q) % q;
    B_source = (myrow + 1 + 2*q) % q;
    MPI_Sendrecv(&tB,1,MPI_DOUBLE,B_dest,982,
                  &ttB,1,MPI_DOUBLE,B_source,982,MPI_col,
                  &mystatus));
    tB = ttB;
    tC += tA*tB; }
```
Coordination Models

• A coordination model describes the interaction between autonomous tasks and deals with communication, synchronization, and task management.
  
  ➢ Focus is on approaches in which the computation and interaction aspects are strictly separated.
  
  ➢ Cooperating tasks are anonymous to each other.

  ▪ Data movement is not coordinated
    ✦ The sender does not require the receiver to know about data movement (puts the data in the right place).
    ✦ The receiver does not require the sender to know about data movement (gets the data from the right place).
Advantages of Coordination Models

- **Separation between computation and coordination**
  - Helps in structuring programs (modularity)
    - Reuse of both computational and coordination modules
    - Modules can interact in multiple / different ways.
    - Modularity in general supports
      - Collaborative software development
      - Verification
      - Maintenance
  - Supports heterogeneity and interoperability
    - Heterogeneity with respect to implementation and software details as well as architectures.
    - Computational modules can work with different interaction modules.
Advantages of Coordination Models [2]

- **Anonymity supports**
  - Dynamic applications
    - Runtime cooperative tasks
  - Code mobility.
    - Code/Data can be constructed to move if necessary
One particular Coordination Model

- **Distributed Shared Memory (DSM)**
  - Physically distributed memory that is used logically as shared memory
    - Hardware or software support for this is required.
- **Common DSM mechanisms**
  - **Page Level**
    - E.g., Kernel extensions to moving virtual memory to other partner “kernels” in addition to Swap space
  - **Data Models**
    - Data Variables or objects are symmetrically identified as logically shared but physically distributed.
      - Objects have methods of access and updates attached 😊
Page Level DSM

- Requires modification of the virtual memory mechanisms.
  - OS source may not be available or modifiable.
    - If not then yet another layer of indirection needed
  - Kernel interaction from one node to another
    - SMP adds yet another layer at each node
  - Consistency model may become complex
    - Is the page change local or global to the SMP image
  - Must move the whole page even if only one variable is needed
    - False sharing at the page level instead of the cache level is a serious performance bottleneck
  - Data movement is “transparent” to the user
Data variable/object mechanisms

• Programmer must identify variables that are shared
  ➢ Not like OpenMP where the default is everything is shared unless specified.

• Operations that are separated among shared and local variables can be tricky
  ➢ Care must be taken to make sure that the consistency is appropriate for the algorithm.

• Data movement is not automatic nor transparent.
  ➢ Either explicit or implicit in the library interface that accesses/moves data
Tuple Based Coordination Models

• A model where the process coordination is via a shared data structure.

• Often looks like a single homogenous entity.

• Multiple processes can access the shared data structure.

• A tuple is an ordered collection of information.
  ➢ Actual details of the data structure are usually hidden from the programmer.

• Tuples are written by a process and read by any other of the cooperating processes.
Global Arrays

- The Global Arrays (GA) toolkit provides a shared memory style programming environment in the context of distributed array data structures
  - So called "global arrays"
  - From the user perspective, a global array can be used as if it was stored in shared memory.
  - All details of the data distribution, addressing, and data access are encapsulated in the global array objects.
  - Information about the actual data distribution and locality can be easily obtained and taken advantage of whenever data locality is important.

- The primary target architectures for which GA was developed are massively-parallel distributed-memory or scalable shared-memory systems.
  - Loosely based on Linda Tuple model.
Global Arrays [2]

- GA divides logically shared data structures into local and remote portions and recognizes variable data transfer costs required to access the data depending on the proximity attributes.
  - A "local" portion of the “logically shared memory” is assumed to be faster to access
  - The remainder ("remote" portion of “logically shared memory”) is considered slower to access.
  - These differences do not hinder the ease-of-use since the library provides uniform access mechanisms for all the shared data regardless where the referenced data is located.
    - Detailed understanding of the distribution can lead to performance improvements.
  - In addition, any processes can access a local portion of the shared data directly/in-place like any other portion of local memory.
    - Access to other portions of the shared data must be done through the GA library calls.

4/8/2005

ComS 425

Spring 2005

120 of 158: Lecture 31
Global Arrays [3]

• GA was designed to complement rather than substitute the message-passing model, and it allows the user/developer to combine shared-memory and message-passing styles of programming in the same program.
  ➢ GA works with MPI, TCGMSG, PVM message passing libraries.

• GA inherits an execution environment from a message-passing library (w.r.t. processes, file descriptors etc.) that started the parallel program.
  ➢ The message-passing library starts the parallel program for GA.

• GA is implemented as a library with C and Fortran-77 bindings
  ➢ Python and C++ interfaces developed but not supported.
Global Arrays [4]

• Explicit library calls are required to use the GA model in a parallel C/Fortran program.
• A disk extension of the Global Array library is supported by its companion library called Disk Resident Arrays (DRA).
  - DRA maintains array objects in secondary storage and allows transfer of data to/from global arrays.
  - We won’t do anything with DRAs in the course 😞
• Basic Functionality
  - The basic shared memory operations supported include
    - get, put, scatter and gather.
    - They are complemented by the atomic read-and-increment, accumulate
      - reduction operations that combines data in local memory with data in the “logically shared memory” locations,
    - lock operations.
Global Arrays [5]

• However, these operations can only be used to access data in global arrays rather than arbitrary memory locations.
  ➢ At least one global array has to be created before data transfer operations can be used.
  ➢ These operations are truly one-sided/unilateral and will complete regardless of actions taken by the remote process(es) that own(s) the referenced data.
  ➢ In particular, GA does not offer or rely on a polling operation or require inserting other library calls on the remote side to assure communication progress.

• A programmer in the GA program has full control over the distribution of global arrays.
  ➢ Both regular and irregular distributions are supported.
Global Arrays [6]

• The GA data transfer operations use an array index-based interface rather than addresses of the shared data.
  ➢ Unlike other systems based on global address space that support remote memory (put/get) operations.
    ▪ E.g., SHMEM
  ➢ GA does not require the user to specify the target process(es) where the referenced shared data resides.
    ▪ It simply provides a global view of the data structures.
    ▪ It “knows” where the data is without you telling it.
  ➢ The higher level array oriented API makes GA easier to use
    ▪ Does not compromising data locality control.
  ➢ The library internally performs global array index-to-address translation and then transfers data between appropriate processes.
Global Arrays [7]

- If necessary, the programmer is always able to inquire:
  - where and an element or array section (e.g., patch) is located, and
  - which process(es) own data in the specified array section.

- The GA toolkit supports three data types:
  - integer,
  - double precision, and
  - double complex.

- The supported array dimensions
  - range from one to seven.
  - This limit follows the Fortran convention.
  - If you are insane, the library can be reconfigured to support more than 7-dimensions but only through the C interface.
Global Arrays Programming Model

• The Global Arrays library supports two programming styles:
 ➢ task-parallel and data-parallel.

• The GA task-parallel model of computations is based on the explicit remote memory copy:
 ➢ The remote portion of shared data has to be copied into the local memory area of a process before it can be used in computations by that process.
 ➢ Of course, the "local" portion of shared data can always be accessed directly thus avoiding the memory copy.

• The data distribution and locality control are provided to the programmer.
 ➢ Data locality information for the shared data is available.
 ➢ Operations for management of its data structures, one-sided data transfer operations, and supportive operations for data locality control and queries.
Global Arrays Programming Model [2]

- The GA shared memory consistency model is a result of a compromise between
  - the ease of use and
  - portable performance.
- The load and store operations are guaranteed to be ordered with respect to each other only if they target overlapping memory locations.
- The store operations (put, scatter) and accumulate complete locally before returning
  - i.e., the data in the user local buffer has been copied out but not necessarily completed at the remote side.
- The memory consistency is only guaranteed for:
  - multiple read operations (as the data does not change),
  - multiple accumulate operations (as addition is commutative), and
  - multiple disjoint put operations (as there is only one writer for each element).
Global Arrays Programming Model [3]

- The application can manage consistency of its data structures in other cases by using:
  - Lock (mutexes),
  - barrier, and
  - fence operations.

- The data-parallel model is supported by a set of (collectively called) functions that operate on global arrays or their portions.

- Underneath, if any interprocessor communication is required, the library uses:
  - remote memory copy or
  - collective message-passing operations.
GA the pictorial view

Shared Object

copy to local memory

local memory

compute/update

local memory

Shared Object

copy to shared object

local memory

Application Guidelines

• **When to use GA:**
  - Algorithmic Considerations
    - applications with dynamic and irregular communication patterns
    - for calculations driven by dynamic load balancing
    - need 1-sided access to shared data structures
    - need high-level operations on distributed arrays for out-of-core array-based algorithms (GA + DRA)
  - Usability Considerations
    - data locality important
    - when coding in message passing becomes too complicated
    - when portable performance is important
    - need object orientation without the overhead of C++
Application Guidelines [2]

• **When not to use GA:**
  
  ➢ **Algorithmic Considerations**
    
    ▪ for systolic communications, or
      
      ♦ Shifting B in Fox’s algorithm is a systolic operation.
    
    ▪ nearest neighbor communications
    
    ▪ regular communication patterns
    
    ▪ when synchronization associated with cooperative point-to-point message passing is needed
      
      ♦ e.g., Cholesky factorization in Scalapack
  
  ➢ **Usability Considerations**
    
    ▪ when interprocedural analysis and compiler parallelization is more effective
    
    ▪ existing language support is sufficient and robust compilers are available!
GA/MA Memory scope SM systems

Physical Memory

Stack

Static Data and Program Instructions

Heap

Shared Memory Segments
GA/MA Memory scope

Physical Memory

Static Data and Program Instructions

Heap

GA

GA

GA

GA

GA

Stack
Fence Operations

• Fence blocks the calling process until all the data transfers corresponding to the GA operations initiated by the calling process complete.

• The typical scenario used is:
  1. Initialize the fence
  2. Global array operations
  3. Fence

• This would guarantee the operations between step 1 and 3 are complete.
Brief Intro to SHMEM

• **One sided Message passing.**
  - Asynchronous data access (like GA)
  - Explicit management of data (like MPI)

• **The logically shared, distributed memory access**

• **SHMEM routines can be used in programs that**
  - perform computations in separate address spaces and
  - that explicitly pass data to and from different processes in the program.

• **SHMEM routines support remote data transfer through**
  - *put operations,*
    - which transfer data to a different PE, and
  - *get operations,*
    - which transfer data from a different PE.
Brief Intro to SHMEM [2]

- **Other operations supported are**
  - broadcast and
  - reduction,
  - barrier synchronization
  - atomic memory operations.

- **Remotely accessible data objects**
  - Typically, target or source arrays that reside on remote processing elements (PEs) are identified by passing the address of the corresponding data object on the local PE.
  - The local existence of a corresponding data object implies that a data object is symmetric.
  - Most uses of SHMEM routines employ symmetric data objects, but on UNICOS/mk systems, another class of data object, asymmetric accessible data objects, can also be passed to SHMEM routines.
Brief Intro to SHMEM [3]

- Symmetric or asymmetric accessible data objects passed to SHMEM routines can be arrays or scalars.
- Accessible addressing classes are as follows:
  - A symmetric data object is one for which the local and remote addresses have a known relationship.
  - You can use SHMEM routines to access remote symmetric data objects by using just the address (pointer).
  - The following are symmetric on UNICOS and UNICOS/mk:
    - Fortran data objects in common blocks or with the SAVE attribute.
    - Non-stack C and C++ variables.
  - The following are symmetric on UNICOS/mk:
    - Fortran arrays allocated with shpalloc(3F)
    - C and C++ data allocated by shmalloc(3C)
    - C and C++ stack variables declared with a #pragma symmetric directive
    - Fortran stack variables declared with a !DIR$ SYMMETRIC directive
Brief Intro to SHMEM [4]

- SHMEM collective routines that operate on the same data object on multiple PEs require that symmetric data objects be passed.
  - This restriction is for algorithm simplicity and efficiency.
  - The SHMEM routines define the set of target PEs by the following triplet of arguments:
    - PE_start, logPE_stride, and PE_size.
- An asymmetric accessible type of data object can be accessed from other PEs through SHMEM routines.
  - An asymmetric accessible object is remotely accessible, but not symmetric.
  - Therefore, these data objects can be accessed through SHMEM routines only if their address is communicated between PEs first by use of a prelude SHMEM or other message passing library call.
Brief Intro to SHMEM [5]

• The following data objects are asymmetric accessible on UNICOS/mk systems:
  ➢ C/C++ data allocated by malloc(3C) and C++ data allocated by the new operator
  ➢ C/C++ variables with automatic or register storage class
  ➢ Fortran arrays allocated with hpalloc(3F)
  ➢ Fortran PE-private data objects on the stack
    ▪ Routine local data.
SHMEM for Fox’s or Cannon’s

- You create the data the same way you would for MPI
- Figure out element/patch of $A$ that is needed and which process that owns it
  - $\text{shmem\_get}(\text{Atmp}, A, \text{Sizeof}A, \text{PE\_owns\_}A)$
- Figure out element/patch of $B$ that would is needed for multiply and process that owns it!
  - $\text{shmem\_get}(\text{Btmp}, B, \text{Sizeof}B, \text{PE\_owns\_}B)$
- Compute contribution
  - $C += \text{Atmp}*\text{Btmp}$
- Rinse, Repeat for all stages.
SMP Based Cluster/MPP

NETWORK

SMP node

SMP node

SMP node

SMP node

4/8/2005

ComS 425

Spring 2005

142 of 158: Lecture 31
Mixed mode programming.

- **Why is this important?**
  - Allows the ability to exploit multiple levels of granularity in the application!!!
  - Naturally maps to current hardware architectures
    - Clusters
    - Large MPPs like the NERSC IBM-SP
- **Why is it hard?**
  - No standards exist for the duality of granularity
  - Hard boundaries are often imposed
  - No fluid transition between levels of granularity
  - Often requires a recursive algorithm to take advantage of the multiple levels of granularity.
An example

- **Fox’s algorithm or Cannon’s algorithm.**
  - Both are distributed memory algorithms for doing matrix multiplication.
  - Both have mechanisms where patches of matrices are moved from one process to another.
    - The details are different for each algorithm.
  - Each patch size leads to a dense matrix multiplication kernel
    - Multiplying the patches on each node.
  - That in most implementations is done using a good serial algorithm.
    - It could be done using a thread based algorithm on an SMP node!
Possible Benefits

• **Reduction of inter-node communications.**
  - **MPI-Only** there are “m” nodes with “k” processors
    - Then m*k processes are involved in the communications
  - **With a Hybrid model**
    - “m” process (one per node) are involved in the inter-node communications
    - “k” threads share the data sent to the node!
Unified Parallel C

- **UPC**

  - An extension to C-99 ANSI standard
    - Extensions are explicit parallelism
    - Familiar syntax and control is high-level or low-level as needed!

  - A distributed shared memory programming model
    - Language based not library based like Global Arrays or SHMEM

  - Current support is limited but available in some fashion on many systems
    - SMP on linux, Solaris, etc.
    - Cluster based on Linux
    - Vendor support
      - Cray, HP

  - Designed to deliver performance with ease of use!
  - Research supported in part by the Programming Models effort. http://www.pmodels.org
DSM Memory Model (processes)

Shared
Global

Private
Local

Process

Process

Process

Process

Process
DSM Memory Model (Threads)
DSM Memory Model (SMP cluster)

Shared Global
Private Local

Process 2 threads
Process 3 threads
UPC execution model

- A collection of threads with independent instruction streams
  - MYTHREAD “macro” is the thread ID
  - Number of threads is the THREADS “macro”
    - Numbered from 0 to THREADS-1
  - SPMD programming model
    - Not as “easy” as OpenMP but much more straightforward than Pthreads!
  - Synchronization mechanisms
  - Memory Consistency mechanisms
    - Different kind of synchronization
  - Cluster based implementations “hide” the runtime exercise of moving data from one “node” to another!
    - Same code runs SMP, Distributed, Cluster of SMP
    - Performance is still an issue!!
UPC Memory Model

- **Variables are**
  - “shared” or
  - Private

- **Pointers are**
  - “to Shared” or private
  - Pointers to shared can point to any address in the shared space
    - Part of the shared space has some “affinity” to a given thread
  - Private pointers can point to:
    - Local addresses or the “shared” space of the thread
  - Dynamic and static allocation of memory is valid for both shared and local addresses
UPC Pointers

• There are essentially 4 kinds of pointers
  ➢ int *ppp;
    ▪ Private pointer pointing to data that is private to the thread
  ➢ shared int *pps;
    ▪ Private pointer pointing to data that is in shared space
  ➢ int *shared psp;
    ▪ Shared pointer pointing to data in thread local space
  ➢ shared int *shared pss;
    ▪ Shared pointer pointing to data in shared space
View of these pointers
Work Sharing with upc_forall

- Upc_forall is a parallel loop work sharing construct
  - Similar to OpenMP omp for or do
  - upc_forall is a collective operation
- The loop syntax is
  
  upc_forall( j = 0 ; j < 1000; j++; affinity)  
  
  {
    work
  }
- The first three components of the forall are identical to the C standard for syntax.
- affinity is either a keyword continue or and expression that is of either type pointer-to-shared or integer
What is the for_all affinity

• If \textit{affinity} is \texttt{continue}
  - Then the loop body is executed on all threads
  - The programmer may control (based on \texttt{MYTHREAD}) which iteration is executed
    - Manual control of concurrency

• If \textit{affinity} is an integer expression
  - The loop body is executed such that the value \texttt{MYTHREAD == (affinity mod THREADS)} controls which iteration is executed by which thread

• If \textit{affinity} is a pointer-to-shared
  - The loop body is executed such that the value \texttt{MYTHREAD==upc_threadof(affinity)} controls which iteration is executed by which thread.
  - An “owner-compute” like construct
Synchronization mechanisms

- UPC offers 2 kinds of barriers, fence operations and locks
- `upc_barrier [integer expression]`
- `upc_fence`
  - Ensures that all shared references issued before the fence are completed before any shared references after the fence are issued
  - Just like the Global Array fence operation!
Synchronization mechanisms [2]

• **A split phase barrier**
  - upc_notify [integer expression]
    - The first half of a barrier
    - Non blocking call
  - upc_wait [integer expression]
    - Second half of a barrier
    - Blocks until all threads make the matching upc_notify call
  - Think about this as the inverse of a critical section
    - There is an implied fence before the notify and after the wait!
UPC Memory Consistency Model

- The consistency model controls the ordering of shared operations in a global context.
  - **Relaxed**
    - Shared operations can be reordered by the compiler and/or runtime system
  - **Strict**
    - Enforces sequential ordering of shared operations
      - All shared data movements complete before new ones begin!
  - Programmers are responsible for using the correct model.
  - Can change during the execution of the code
  - Specified via
    - Declarations
    -Pragma statements
    - Use of barriers, fences, global operations etc.