CH1 Parallel Computers

-----------------------Parallel Computers-----------------------

Share Memory Multiprocessor System: unique address

Message-Passing Multicomputer: cannot access other M

Distributed Shared Memory: still need message passing

MIMD: each processor has a separate program, and one instruction stream is generated from each program for each processor. Each instruction operates upon different data. Both share memory and message-passing multiprocessors are MIMD

SIMD: Each processor executes the same instruction in synchronism, but using different data.

MPMD: cluster, similar to MIMD. SPMD: similar to SIMD

--------------------------------Network--------------------------------

Communication latency: total time to send message. network latency + message latency.

Message latency: time to send a zero-length message.

Diameter: minimum number of links between two farthest nodes in the network.

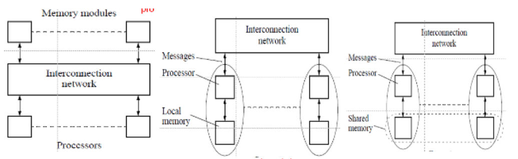
Bisection width: number of link that must be cut to divide network into two equal parts.

Circuit switching:系统间需要预约传输线路才可以进行持续通信。Uninterrupted. All links are reserved for the transfer until message transfer is complete.

Packet switching: packet switch是不需要预约的，每个connection是竞争关系，线路资源先到先得。source将要传输的长信息分成若干packet,传输过程中按数据包传输.发送数据包的第一个比特之前, switch必须已经接收到完整的一个包

Virtual cut-through: if outgoing link is available, message is immediately passed forward without stored. If path is block. Message needs to be stored.

Wormhole routing: When the head is moving through the link (not reaching destination yet), the link of each node that had visited by the head will be occupied. Occupied link will be released when the last flit go through that link.



--------------------------------Embedding--------------------------------

Definition: Describes mapping nodes of one network onto another network.

Dilation: used to indicate the quality of the embedding. $\frac{Max(links, embeding)}{Max(links, embedded)}$.

Perfect embedding line/ring into mesh/torus or mesh onto hypercube, have dilation of 1.

Super-linear Speedup: Reason: extra memory in the multiprocessor system which hold more of problem data at any instant. Happen in search algorithms.

CH2 Message Passing Computing

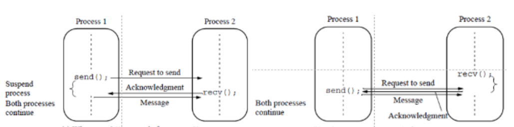
------------------------------Programming Options------------------------------

1: designing a special parallel programming language.

2: extending the syntax words of an existing high-level language

3: using existing sequential high-level language and providing a library of external procedures for message passing

Synchronous Message Passing:



MPI Blocking: return after their local actions complete, though the message transfer may not have been complete. **MPI Non-blocking:** return immediately.

----------------------Parallel Virtual Machine----------------------

Basic Message-Passing Routine: all PVM send routines are nonblocking. PVM receive routines can be blocking or nonblocking. Data packed into a s_buffer prior to sending data. Receiving process must unpack its r_buffer

--------------------------------Blocking Routines--------------------------------

Return when they are locally completed – when the location used to hold the message can be used again or altered without affecting the message being sent. A blocking send will send the message and return. This does not mean that the message has been received, just that process is free to move

----------------------Non-blocking Routines----------------------

MPI_Isend(), MPI_Irecv(), will return immediately even before source location is safe to be altered; or no message to accept.

----------------------Message Passing Interface(MPI)----------------------

Global & Local Variable: Any global declarations of variable will be duplicated in each process. Local: if(process_num == a)

Safe Message Passing: Use communicators, prevent message pass interleave

Intracommunicator: for communication within a group

Intercommunicator: for communication between groups

MPI_Send(buf, count, datatype, dest, tag, comm)

MPI_Recv(buf, count, datatype, src, tag, comm, status)

Send Communication Modes:

1. Standard Mode Send: Not assumed that corresponding receive routine has started. Amount of buffering not defined by MPI. If buffering is provided, send could complete before receive reached.
2. Buffered Mode: Send may start and return before a matching receive. Necessary to specify buffer space via routine: MPI_Buffer_attach()/detach
3. Synchronous Mode: Send & Receive can start before each other but can only complete together
4. Ready Mode: Send can only start if matching receive already reached.

----------------------MPI Time Complexity----------------------

Parallel Execution Time: $t_p = t_{comp} + t_{comm}$

Communication Time: $t_{comm} = t_{startup} + nt_{data}$

Important Notice: t_p normalized in units of an arithmetic operation, which will depend upon computer system. Computation requires m computational steps so that $t_{comp} = m.\ t_{comm} = q(t_{startup} + mt_{data})$ q messages

Cost = $t_p \times n = k \times t_s$

Latency Hiding: 1. overlap communication with subsequent computations. 2. mapping multiple processes on a processor and use a time-sharing facility that switches for one process to another when the first process is stalled by passing.

CH3 Embarrassingly Parallel Computations

An embarrassingly parallel workload or problem (also called perfectly parallel or pleasingly parallel) is one where little or no effort is needed to separate the problem into a number of parallel tasks.

Image Shift:

Sequential: t_s=n^ $t_{comm} = p(t_{startup} + 2t_{data}) + 4n^2(t_{startup} + t_{data}) = O(p + n^2)$

Communication:

Computation: t_compu=2(n^2/p)=O(n^2/p)

Parallel: t_p=t_commu+t_compu

Mandelbrot Steps:

1. Communication: send data $t_{comm} = s(t_{startup} + t_{data})$
2. Computation: $t_{comp} \le (\max * n)/s$
3. Communication: gather results: $t_{comm} = n(t_{startup} + t_{data})/s$

Random Number Generation:

Evaluating $x_{i+1} = (ax_i + c) mod\ m$

$x_{i+k} = (Ax_i + C) mod\ m$ where $A = a^k mod\ m, C = c(a^0 + .. a^{k-1})$modm

CH4 Partitioning & Divide-and-Conquer Strategies

Sum *n* number: seq O(n)

1. Communication: $t_{comm1} = m(t_{startup} + \left(\frac{n}{m}\right)t_{data})$
2. Computation: $t_{comp1} = \frac{n}{m} - 1$
3. Communication: return result: $t_{comm2} = m(t_{startup} + t_{data})$
4. Computation: sum $t_{comp2} = m - 1$ Overall is O(m+n)

Execution time: $t_p = \frac{n(p-1)}{p} t_{data} + t_{data} \log p + \frac{n}{p} + \log p$

Bucket Sort:

Seq: $t_s = n + n log\left(\frac{n}{m}\right) = O(n log\ (n/m))$

Para: $t_p = n + \left(\frac{n}{p}\right) \log\left(\frac{n}{p}\right) = O\left(\left(\frac{n}{p}\right) \log\left(\frac{n}{p}\right)\right)$

t_p

$= t_{startup} + t_{data} n + \frac{n}{p} + (p-1)\left(t_{startup} + \left(\frac{n}{p^2}\right) t_{data}\right) + \left(\frac{n}{p}\right) \log\left(\frac{n}{p}\right)$

CH5 Pipelined Computations

Def: the problem is divided into a series of tasks that have to be completed one after the other, each task will be executed by a separate process or processor.

Three Types of Pipelined Computations that can improve speed:

1. If more than one instance of the complete problem is to be executed
2. If a series of items must be processed, each require multiple operations
3. If info to start the next process can be passed forward before the process has complete all its internal operations.

Execution Time:

Type1: $t_{total} = (time\ for\ 1\ pipeline\ cycle)(number\ of\ cycles) = (t_{comp} + t_{comm})(m + p - 1)$ where m instances of problem, p pipeline stages. $t_a = \frac{t_{total}}{m}$

Single Instance of Problem: $t_{comp} = 1, t_{comm} = 2(t_{startup} + t_{data}), t_{total} = \left(2(t_{startup} + t_{data})\right)n$. Time Complexity=O(n)

Multiple instances: $t_a = \frac{t_{total}}{m} \approx 2(t_{startup} + t_{data}) + 1$

Sorting number (insertion sort): seq: t_s=n(n-1)/2, **parallel:** t_total=(1+2(t_startup+t_data))(2n-1)

CH6 Synchronous Computation

Def: In a synchronous application, all processes synchronized at regular points. ***Barrier*** synchronizing point.

MPI: MPI_Barrier(), called by each process, block until all member arrive.

PVM: has unusual feture of specifying the # of process arrive that continue.

Centralized Counter Implementation(Linear Barrier), two phases:

1) A process enter *arrival phase and wait others. 2)go to departure phase.*

A Barrier might used more than once

MPI prevent deadlocks: by MPI_Sendrecv() and MPI_Sendrecv_replace()

Prefix Sum: compute all partial summation of a list of num. Seq O(n^2)

Solving Linear Equation: *t_comp=n/p(2n+4)torque; t_comm=p(t_startup+(n/p)t_data)torque=(pt_startup+nt_data)torque*

Heat Distribution: Block Parti: t_comm=8(t_startup+sqrt(n/p)t_data) ***Strip partition:*** t_comm=4(t_startup+sqrt(n)t_data) when partition, p, >=9, block better.

CH7-Load Balancing and Termination Detection

Load balancing – used to distribute computations fairly across processors in order to obtain the highest possible execution speed.

Termination Detection – detecting when a computation has been completed. Difficult when distributed.

Static Load Balancing (SLB):

SLB techniques:

Round robin algorithm; *take tasks in seq, circular order*

Randomized algorithm; *take tasks by random select*

Recursive bisection; *recursively divide to equal parts, min(comm)*

Simulated annealing; *an optimization technique*

Genetic algorithm; *another optimization technique*

SLB techniques drawback:

1) Very difficult to estimate accurately the execution times of various parts of a program without actually executing them.

2) Communication delays that vary under different circumstances.

3) Some problems have an indeterminate number of steps to reach their solution

Processes & Processors

Computation will be divided into tasks, and processes perform the tasks. Processes are mapped onto processors.

Dynamic Load Balancing (DLB):

Centralized DLB

Tasks are handed out from a centralized location, where a clear master-slave structure: *master hold the collection of tasks, slaves process task and ask more when it's done.*

Termination of Centralized DLB:

The task queue is empty and every process has made a request for another task without any new tasks being generated.

Decentralized DLB

Tasks are passed between arbitrary processes. A collection of worker processes operate upon the problem and interact among themselves, finally reporting to a single process. A worker process may receive/send tasks from/to other worker processes.

Task Transfer Mechanism

Receiver-Initiated Method

A process requests from other processes when it has few or no tasks to perform. This method work well at high system load. However its expensive to determine process loads.

Sender-Initiated Method

A heavy load process sends tasks to others who accept them. This method work well for light overall system loads. Also, expensive to check process load.

Load Balancing Structure

Line Structure

Master Process: feeds the queue with tasks at one end, tasks are shifted down the queue.

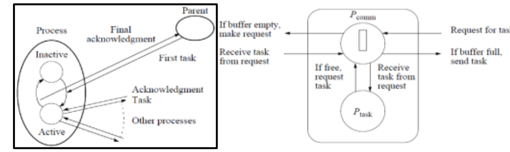
Worker Process: detects tasks at its input from the queue and take it if worker is idle.

Shifting Actions:

Tree Structure

Tasks passed from node into one of the two nodes below it when node buffer empty.

*nonblockcing receive, MPI_Irecv(), posts a request for message and returns immediately.



Distributed Termination Detections

Termination Conditions

At time *t* requires these conditions:

1) Application-specific local termination conditions exist throughout the collection of processes, at time *t*.

2) There are no messages in transit between processes at time *t*

Using Acknowledgment Messages (figure in black box)

On every occasion when process receives a task, it immediately sends an acknowledgement message, except if the process it receives the task from is its parent process.

It only sends an acknowledgment message to its parent when it is ready to become inactive, i.e. when:

1) Its local termination condition exists

2) It has transmitted all its acknowledgments for tasks received

3) It has received all its acknowledgments for tasks it sent out

Termination Algorithms

Ring Termination Algorithm:

Dscb: Pass a termination token in ring to ensure everyone is done

It assumes that a process cannot be reactivated after reaching its local termination conditions.

It does not apply to work pool problems in which a process can pass a new task to an idle process.

Dual-Pass Ring Termination Algorithm:

Dscb: if reactivated, pass another token through one round

Receive black token: global termination may not have occurred and the token must be recirculated around again.

Tree Algorithm:

Dscb: from leaf to root, up to the point of termination.

Fixed Energy Distributed Termination Algorithm:

Dscb: Sub-processes handle tasks with energy. When a process is idle, it pass energy back before requesting new task. A process will not hand back energy until all the energy it handed out is returned and combined to the total. When all energy back to the root, and root become idle, this is a termination.

Significant Disadvantage: Dividing the energy well be finite precision and adding the partial energies may not equate to the original energy.

CH8-Programming with Shared Memory

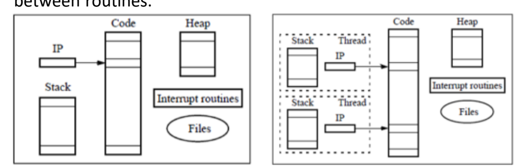
Shared Memory Multi-processor (SMMP) System

A *single address space* exists, meaning that each memory location is given a unique address within a single range of addresses.

Processes & Threads

"Heavy weight" process: complete separate program with its own variables, stack, and memory allocation.

Threads: shares the same memory space and global variables between routines.



Detached Threads

Threads that are not joined are called *detached threads*.

Statement Execution Order

On a multiprocessor system, instructions of individual processes/threads might be interleaved. Compilers will reorder the codes to improve performance.

Thread-Safe Routines

System calls for library routines are called *thread safe* if they can be called from multiple threads simultaneously and always produce correct result.

Critical Section

Definition

A mechanism for ensuring that only one process accesses a particular resource at a time.

Mutual Exclusion

Once the process has finished its critical section, another process is allowed to enter a critical section for the same resource.

Pthread Lock Routines

Locks are implemented in Pthread with *mutually exclusive lock* variables, or *"mutex"* variables. System will select one thread to proceed if multiple thread waiting for a single mutex.

Deadlock
Occur with multiple process when on requires a resource held by others mutually. Deadlock can be eliminated between *n* processes accessing resource than *n*.
Pthread has *pthread_mutex_trylock()* to prevent deadlocks.

Semaphores

A semaphore, s, is a positive interger (include 0) operated upon by two operations:
P Operation: waits until s is greater than 0 and then decrements s by 1 and allows process to continue.
V Operation: increments s by 1 to release one of the waiting processes (if any)
Notice: these operations are performed indivisibly. s can take on positive values other than zero and one, meaning of recording the number of "resource unites" available or used and can be used to solve producer/consumer problems. Semaphore routines exist for UNIX processes, though they can be written and they do exist in real-time extension to Pthreads.

Monitor

A suit of procedures that provides the only method to access a shared resource. Reading and writing can only be done by using a monitor procedure, and only one process can use a monitor at any instant.

Condition Variables

A critical section is to be executed if a specific global condition exist. With locks the global variable would need to be examined at frequent intervals within a critical section, which is time-consuming. This can be solve by *condition variable*

Bernstein's Condition

Condition that sufficient to determine whether two processes can be executed simultaneously:
I_i *is the set of memory locations read by process* P_i
O_j *is the set of memory locations altered by process* P_j
For two process P_1 and P_2 to be execute simultaneously:
$I_1 \cap O_2 = \emptyset$
$I_2 \cap O_1 = \emptyset$
The set of outputs of each process must also be different:
$O_1 \cap O_2 = \emptyset$

Shared Data in Systems with Caches

Cache Coherence Protocols: In the *update policy*, copies of data in all caches are updated at the time one copy is altered.
In the *invalidate policy*, when one copy of data is altered, the same data in any other cache is invalidated. These copies are only updated when the associated processor makes reference for it.
False Sharing: Different parts of block required by different processors but not same bytes. If one processor writes to one part of the block, copies of the complete block in other caches must be updated or invalidated though the actual data is not shared.
Compiler to alter the layout of the data stored in the main memory, separating data only altered by one processor into different blocks.

Quinn 17

Shared-memory Model: #active threads 1 at start and finish of the program, changes dynamically during execution. Execute and profile sequential program. Incrementally make it parallel. Stop when further effort not warranted.
Message-passing Model: all processes active through execution. Sequential-to-parallel transformation requires major effort. Transformation done in one giant step.
Incremental Parallelization:
Sequential program a special case of a share-memory parallel program. Parallel share-memory programs may only have a single parallel loop. Incremental Parallel: process of converting a sequential program to a parallel program a little bit at a time.
Execution Context: address space containing all of the variables a thread may access, includes: static variables, dynamically allocated data structures in the heap, variables on run-time stack, additional run-time stack for funcs invoked by thread.
Shared/private variables: same address in threads or not.
Omp_get_num_procs: return #physical processors
Private clause: direct compiler to make some variables private
Firstprivate clause: create private variables having initial values identical to the variable controlled by the master thread as the loop is entered.
Critical pragma: a portion of code that only thread at a time.
Source of Inefficiency: Update to area inside a critical section.
Reduction (<op> :<variable>)

Performance Improvement#1: Too many fork/join lower perform. Inverting loops may help performance if: parallelism is in inner loop. After inversion, the outer loop can be made parallel. Inversion does not significantly lower cache hit rate.
Improvement#2: If loop has too few iterations, for/join overhead is greater than time savings from parallel execution.
#pragma omp parallel for if(n > 500)
Improvement#3: **Schedule** clause to specify how iteration of a loop should be allocated to threads; static/dynamic inside loop.
Static/Dynamic Scheduling: low/high overhead, imbalance/not
Chunk: is a contiguous range of iteration. Increasing chunk size reduces overhead and may increase cache hit rate. Decreasing chunk size allows finer balancing of workloads.
#schedule (<type>[,<chunk>]). Allowed types:static/ dynamic/ guided/runtime/
Functions for SPMD-style Programming:
Omp_get_thread_num: return ID
Omp_get_num_threads: return total active num of threads
#pragma omp single: only a single process, no matter who.
Nowait clause: a barrier synchronization at end of every parallel for statement.
#pragma omp parallel sections
#pragma omp section
some code here, pack in a section

Characteristic	*OpenMP*	*MPI*
Suitable for multiprocessors	Yes	Yes
Suitable for multicomputers	No	Yes
Supports incremental parallelization	Yes	No
Minimal extra code	Yes	No
Explicit control of memory hierarchy	No	Yes

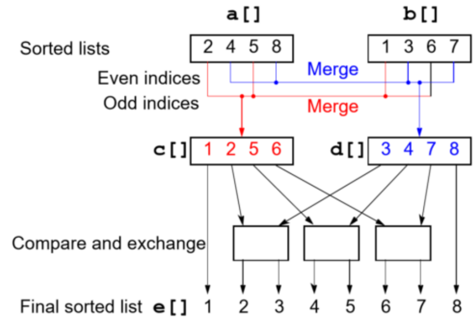
Quinn 18

Why C+MPI+OpenMP faster:
1. Lower communication overhead. 2. more portion of program may be practical to parallelize. 3. May allow more overlap of communications with computations

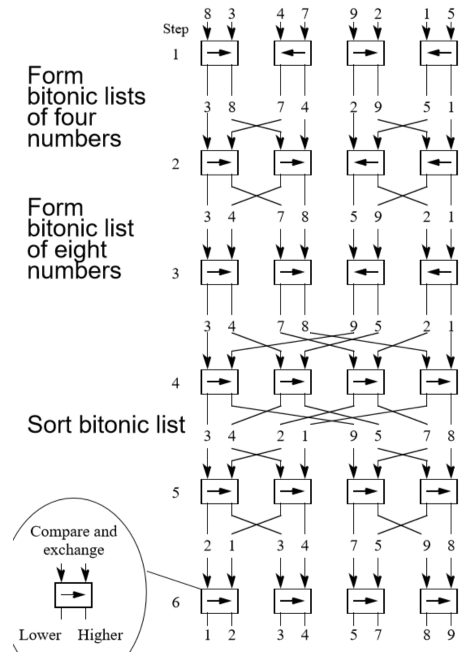
Algorithms and Sorting

Sorting: O(nlogn) optimal for any sequential sorting algorithm. O(logn) optimal for parallel.
1. Compare & exchange: 1.P1 send A to P2, P2, compare A with B. 2. both send, both do compare. Notice that different precision could conceivably produce different answers.
2. Bubble sort: largest number move to end of list by comparisons. Number of operation: n(n-1)/2, O(n^2); odd even
3. merge sort: seq O(nlogn), para 2logn steps, each step more than 1 operation.
4. Quick sort: seq O(nlogn), choose pivot. May not balance in parallel. Pivot selection is critical

Batcher's parallel sorting: Odd-even merge sort & bitonic mergesort. Both well balanced and have para O(log^2 n) with n processors
Odd-even merging of two sorted lists
Bitonic Sequence
A monotonic increasing sequence is a sequence of increasing numbers.
A bitonic -> one increasing and one decreasing.



Feature: If we perform a compare-and-exchange operation on ai with ai+n/2 for all i , where there are n numbers in the sequence, get TWO bitonic sequences, where the numbers in one sequence are all less than the numbers in the other sequence
Sorting a bitonic sequence



#steps: n=2^k,k phases, steps=logn(logn+1)/2 = O(log^2n)
Summary: odd-even transposition: O(n). Parallel merge: O(n) but unbalanced. Parallel quicksort:O(n), unbalanced, can be O(n^2). Odd-even mergesort and bitonic merge: O(log^2 n)
Sorting on specific network: Algorithms can take advantage of the underlying interconnection network of the parallel computer. Two network structures have received specific attention: the mesh and hypercube because parallel computers have been built with these networks.
MPI does provide features for mapping algorithms onto meshes, and one can always use a mesh or hypercube algorithm even if the underlying architecture is not the same.
Shear sort: sqrt(n)(logn+1) for n number, on sqrt(n)sqrt(n) mesh
Rank sort: seq >O(nlogn), para O(logn) with n^2 processors, O(n) with n processors.
Counting sort: a stable sorting algorithm. requires O(logn) time with n − 1 processors. The final sorting stage can be achieved in O(n/p) time with p processors or O(1) with n processors.
Radix sort: Already mentioned parallelized counting sort using prefix sum calculation, which leads to O(logn) time with n − 1 processors and constant b and r.