Overview of Parallel Programming

May 2012

Indra Mani

© 2009 IBM Corporation

IBM

Agenda

- Section 1: Overview
- Section 2: Basic Terminology
- Section 3: Concepts and Techniques
- Section 4: Examples
- Discussion

© 2009 IBM Corporation

Overview

© 2009 IBM Corporation

What is parallel programming

Traditionally, software has been written for serial computation:

- To be run on a single computer having a single Central Processing Unit (CPU);
- A problem is broken into a discrete series of instructions.
- Instructions are executed one after another.
- Only one instruction may execute at any moment in time.

IBM CONFIDENTIAL
© 2009 IBM Corporation

What is parallel programming ...

In the simplest sense, parallel computing is the simultaneous use of multiple compute resources to solve a computational problem:

- To be run using multiple CPUs
- A problem is broken into discrete parts that can be solved concurrently
- Each part is further broken down to a series of instructions
- Instructions from each part execute simultaneously on different CPUs

© 2009 IBM Corporation

What is parallel programming ...

The compute resources might be:

- A single computer with multiple processors;
- An arbitrary number of computers connected by a network;
- A combination of both.

© 2009 IBM Corporation

What is parallel programming ...

The computational problem should be able to:

- Be broken apart into discrete pieces of work that can be solved simultaneously;
- Execute multiple program instructions at any moment in time;
- Be solved in less time with multiple compute resources than with a single compute resource.

© 2009 IBM Corporation

Limits to serial computing

- Transmission speeds
- Limits to miniaturization
- Economic limitations
- Hardware level parallelism evolved to improve performance
  - Multiple execution units
  - Pipelined instructions
  - Multi-core

IBM CONFIDENTIAL

© 2009 IBM Corporation

Why Use Parallel Computing?

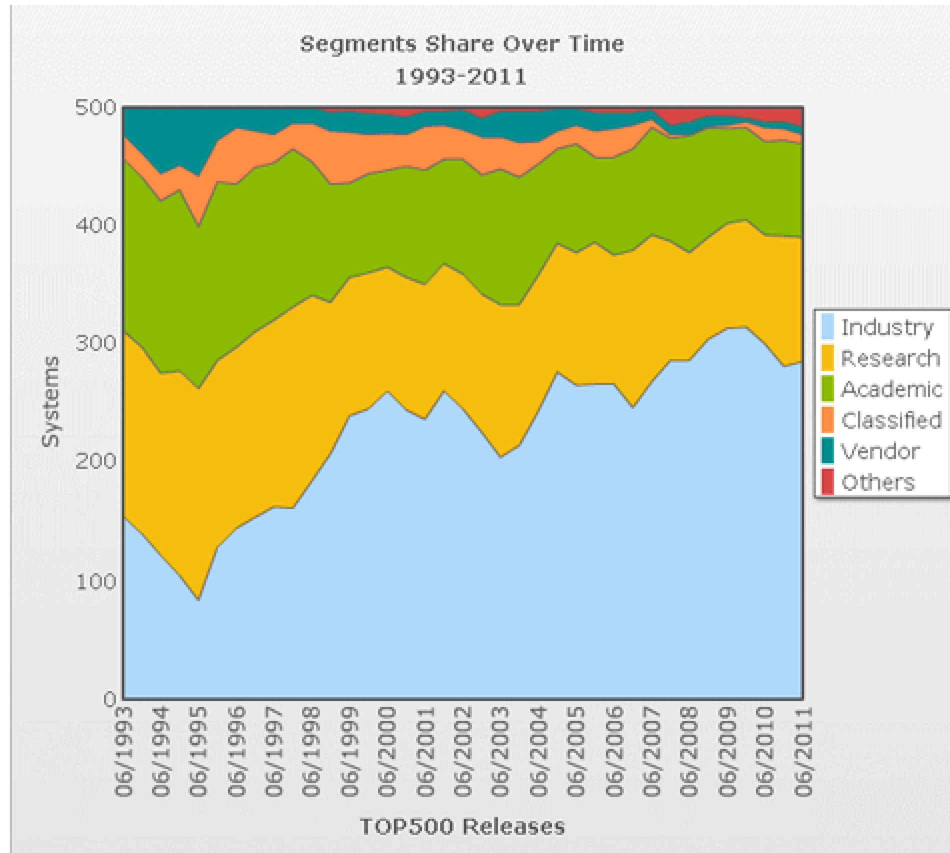
- Reducing time to solve a problem.
- Increasing quality of the solution.
- Solve larger problems.
- Provide concurrency.
- Use of non-local resources.

IBM CONFIDENTIAL

© 2009 IBM Corporation

Who are using parallel computing?

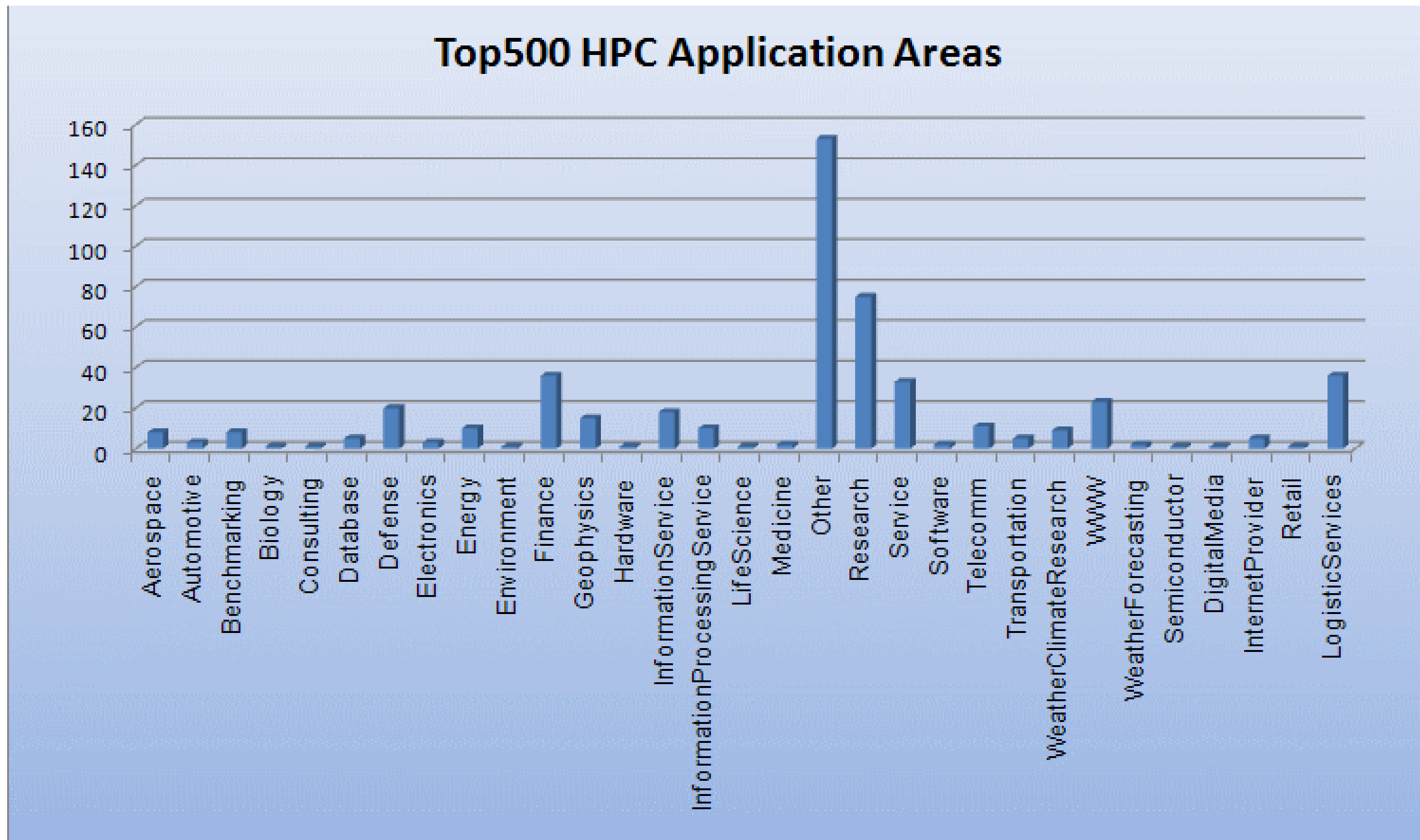
- Top500.org provides statistics on parallel computing



© 2009 IBM Corporation

Application areas for parallel computing.

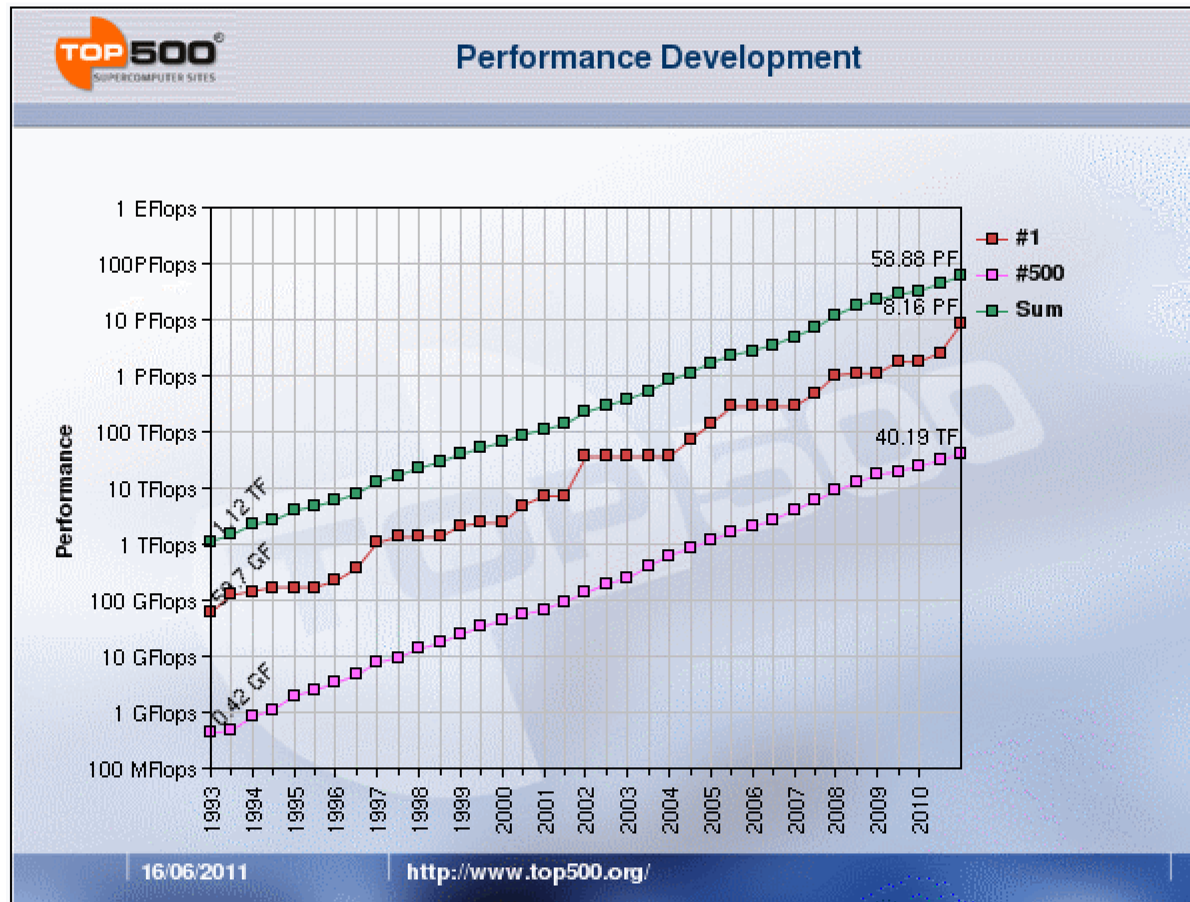
- Top500.org provides statistics on parallel computing



© 2009 IBM Corporation

Trends in performance of parallel computing

- Top500.org provides statistics on parallel computing



© 2009 IBM Corporation

Future

- During the past 20+ years, the trends indicated by ever faster networks, distributed systems, and multi-processor computer architectures (even at the desktop level) clearly show that parallelism is the future of computing.

- In this same time period, there has been a greater than 1000x increase in supercomputer performance, with no end currently in sight.

- The race is already on for Exascale Computing!

© 2009 IBM Corporation

Basic Terminology

© 2009 IBM Corporation

Flynn's Classical Taxonomy

There are different ways to classify parallel computers. One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.

S I S D Single Instruction, Single Data	S I M D Single Instruction, Multiple Data
M I S D Multiple Instruction, Single Data	M I M D Multiple Instruction, Multiple Data

© 2009 IBM Corporation

Some General Parallel Terminology

Supercomputing / High Performance Computing (HPC)

- **Node** - A standalone "computer in a box".
- **CPU / Socket / Processor / Core**
- **Task** - A logically discrete section of computational work. A parallel program consists of multiple tasks running on multiple processors.
- **Communication**
- **Synchronization**

© 2009 IBM Corporation

Some General Parallel Terminology ...

- **Granularity** - a qualitative measure of the ratio of computation to communication.
  - Coarse: relatively large amounts of computational work are done between communication events
  - Fine: relatively small amounts of computational work are done between communication events
- **Observed Speedup** - One of the simplest and most widely used indicators for a parallel program's performance.
  - Speedup = wall-clock time of serial execution / wall-clock time of parallel execution

IBM CONFIDENTIAL

© 2009 IBM Corporation

Some General Parallel Terminology ...

- **Parallel Overhead** - The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:
  - Task start-up time
  - Synchronizations
  - Data communications
  - Software overhead imposed by parallel compilers, libraries, tools, OS, etc.
  - Task termination time

© 2009 IBM Corporation

Some General Parallel Terminology ...

- **Scalability** - Refers to a parallel system's ability to demonstrate a proportionate increase in parallel speedup with the addition of more processors. Factors that contribute to scalability include:
  - Hardware - particularly memory-cpu bandwidths and network communications
  - Application algorithm
  - Parallel overhead related
  - Characteristics of your specific application and coding

© 2009 IBM Corporation

Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory (without threads)
  - Threads (pthreads, OpenMP)
  - Distributed Memory / Message Passing (MPI)
  - Data Parallel (HPF, X10)
  - Hybrid (for eg. MPI+OpenMP+OpenCL)
  - Single Program Multiple Data (OpenCL,CUDA)
  - Multiple Program Multiple Data

- Parallel programming models exist as an abstraction above hardware and memory architectures.

© 2009 IBM Corporation

OpenMP

- Compiler directive based; can use serial code
- Jointly defined and endorsed by a group of major computer hardware and software vendors. The OpenMP Fortran API was released October 28, 1997. The C/C++ API was released in late 1998.
- Portable / multi-platform, including Unix and Windows NT platforms
- Available in C/C++ and Fortran implementations
- Can be very easy and simple to use - provides for "incremental parallelism"

© 2009 IBM Corporation

MPI

- Message passing implementations usually comprise a library of subroutines. Calls to these subroutines are imbedded in source code.
- The programmer is responsible for determining all parallelism.
- Historically, a variety of message passing libraries have been available since the 1980s.
- These implementations differed substantially from each other making it difficult for programmers to develop portable applications.
- In 1992, the MPI Forum was formed with the primary goal of establishing a standard interface for message passing implementations.

© 2009 IBM Corporation

MPI ...

- Part 1 of the Message Passing Interface (MPI) was released in 1994. Part 2 (MPI-2) was released in 1996.
- MPI is now the "de facto" industry standard for message passing, replacing virtually all other message passing implementations used for production work.
- MPI implementations exist for virtually all popular parallel computing platforms.
- Not all implementations include everything in both MPI1 and MPI2.
- Popular MPI implementation include MPICH2, LAM/MPI, Open MPI

© 2009 IBM Corporation

Concepts and Techniques

© 2009 IBM Corporation

Limits of Parallel Programming

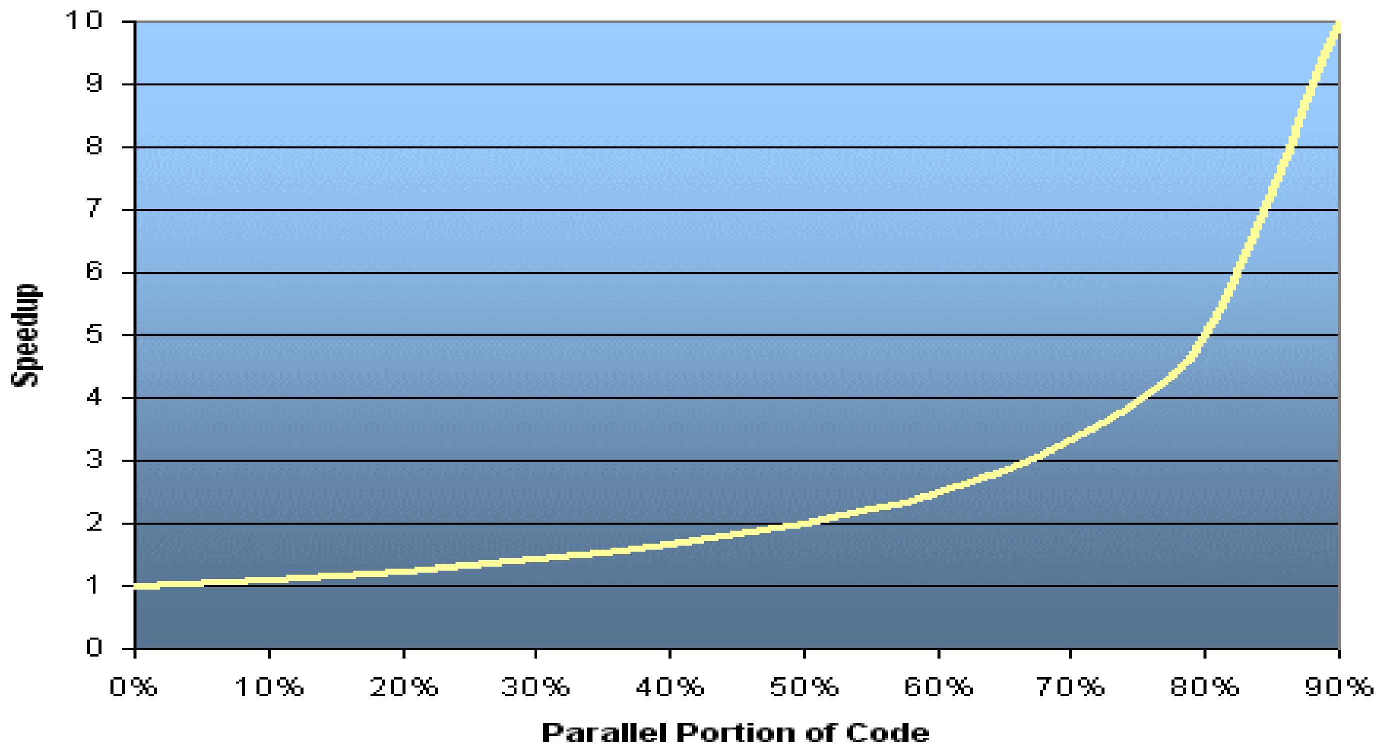
Amdahl's Law:

- Amdahl's Law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:
  - Max speedup = 1 / (1 – P)
  - If none of the code can be parallelized, P = 0 and the speedup = 1 (no speedup).
  - If all of the code is parallelized, P = 1 and the speedup is infinite (in theory).
  - If 50% of the code can be parallelized, maximum speedup = 2, meaning the code will run twice as fast.
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:
  - Speedup = 1 / ((P/N + S))
  - where P = parallel fraction, N = number of processors and S = serial fraction.

IBM CONFIDENTIAL
© 2009 IBM Corporation

Limits of Parallel Programming ...

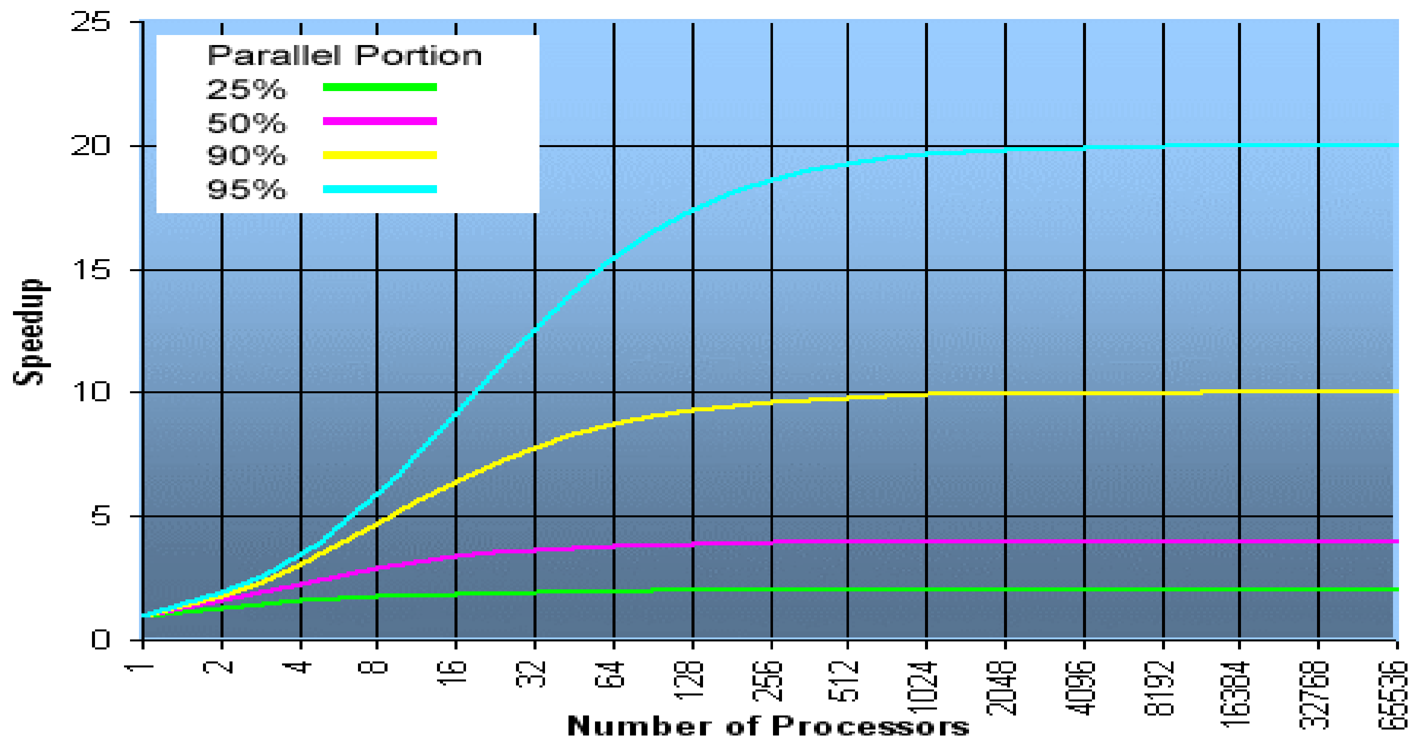
Max possible speedup against parallelizable fraction of code:



© 2009 IBM Corporation

Limits of Parallel Programming ...

possible speedup with increasing number of processors



© 2009 IBM Corporation

Understand the Problem and the Program

» determine whether or not the problem is one that can actually be parallelized:

» Example of Parallelizable Problem:

– Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

– This problem is able to be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

» Example of a Non-parallelizable Problem:

– Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula: F(n) = F(n-1) + F(n-2)

» This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones.

IBM CONFIDENTIAL
© 2009 IBM Corporation

Understand the Problem and the Program ...

Identify inhibitors to parallelism:

- One common class of inhibitor is data dependence, as demonstrated by the Fibonacci sequence above.

Identify the program's hotspots:

- Know where most of the real work is being done. The majority of scientific and technical programs usually accomplish most of their work in a few places.
- Profilers and performance analysis tools can help here
- Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

IBM CONFIDENTIAL

© 2009 IBM Corporation

Understand the Problem and the Program ...

Identify bottlenecks in the program:

- Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
- May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas

Investigate other algorithms if possible:

- This may be the single most important consideration when designing a parallel application.

IBM CONFIDENTIAL
© 2009 IBM Corporation

Partitioning

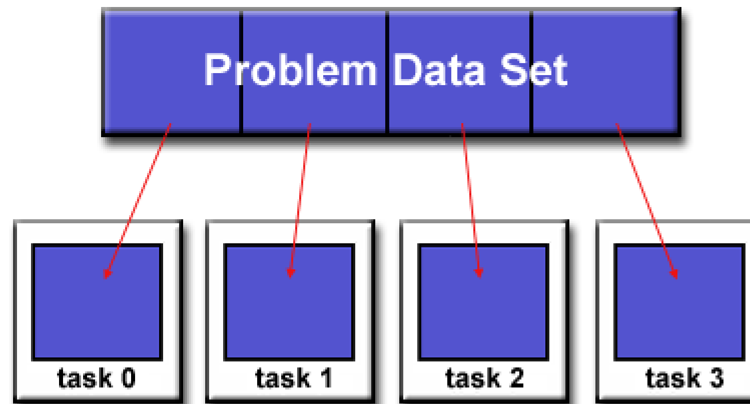
- One of the first steps in designing a parallel program is to break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.
- There are two basic ways to partition computational work among parallel tasks:
  - domain decomposition.
  - functional decomposition.

© 2009 IBM Corporation

Partitioning ...

Domain Decomposition:

- In this type of partitioning, the data associated with a problem is decomposed. Each parallel task then works on a portion of of the data.



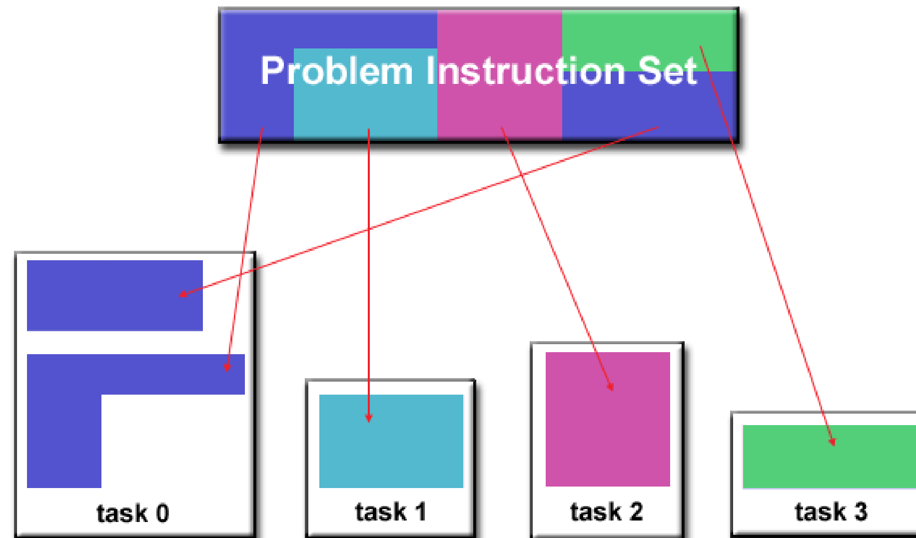
- There are different ways to partition data:
  - Block
  - cyclic

© 2009 IBM Corporation

Partitioning ...

Functional Decomposition:

- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.



© 2009 IBM Corporation

Comminication

The need for communications between tasks depends upon your problem:

- You DON'T need communications
  - Some types of problems can be decomposed and executed in parallel with virtually no need for tasks to share data.
  - For example, an image processing operation where every pixel in a black and white image needs to have its color reversed.
  - These types of problems are often called embarrassingly parallel because they are so straight-forward. Very little inter-task communication is required.
- You DO need communications
  - Most parallel applications are not quite so simple, and do require tasks to share data with each other.
  - For example, a 3-D heat diffusion problem requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.

© 2009 IBM Corporation

Synchronization

Types of Synchronization:

- Barrier
  - Each task performs its work until it reaches the barrier. It then stops, or "blocks".
  - When the last task reaches the barrier, all tasks are synchronized.
- Lock / semaphore
  - Typically used to serialize (protect) access to global data or a section of code. Only one task at a time may use (own) the lock / semaphore / flag.
  - Can be blocking or non-blocking
- Synchronous communication operations
  - Involves only those tasks executing a communication operation
  - When a task performs a communication operation, some form of coordination is required with the other task(s) participating in the communication.
  - For example, before a task can perform a send operation, it must first receive an acknowledgment from the receiving task that it is OK to send.

© 2009 IBM Corporation

Examples

© 2009 IBM Corporation

Vectoradd – an OpenMP Example

- Serial code snippet
  - ...
  - for (i = 0; i < maxSize; i++) {
  - C[i] = A[i] + B[i];
  - }
  - // use C after this.
- Parallelization using OpenMP pragmas
  - ...
  - #pragma omp parallel for
  - for (i = 0; i < maxSize; i++) {
  - C[i] = A[i] + B[i];
  - }
  - #pragma omp barrier
  - // use C after this.

© 2009 IBM Corporation

2-D Array Processing – an MPI Example

This example demonstrates calculations on 2-dimensional array elements, with the computation on each array element being independent from other array elements.

- The serial program calculates one element at a time in sequential order.
- Serial code could be of the form:

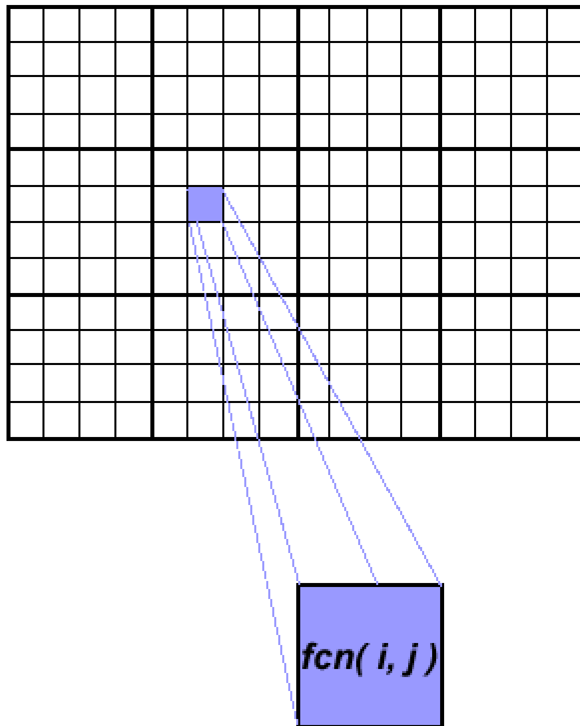
```
do j = 1,n
do i = 1,n
  a(i,j) = fcn(i,j)
end do
end do
```

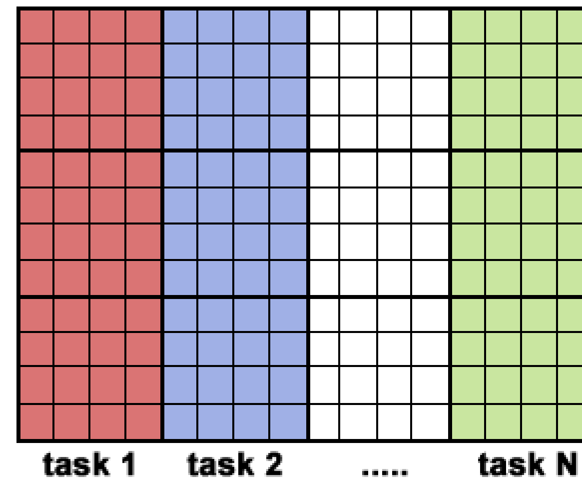
- The calculation of elements is independent of one another - leads to an embarrassingly parallel situation.
- The problem should be computationally intensive.

© 2009 IBM Corporation

2-D Array Processing – an MPI Example ...

Data layouts for serial and parallel case





© 2009 IBM Corporation

2-D Array Processing – an MPI Example ...

- Arrays elements are distributed so that each processor owns a portion of an array (subarray).
- Independent calculation of array elements ensures there is no need for communication between tasks.
- Distribution scheme is chosen by other criteria, e.g. unit stride (stride of 1) through the subarrays. Unit stride maximizes cache/memory usage.
- Since it is desirable to have unit stride through the subarrays, the choice of a distribution scheme depends on the programming language.
- After the array is distributed, each task executes the portion of the loop corresponding to the data it owns. For example, with Fortran block distribution:
  - do j = mystart, myend
  - do i = 1,n
  - a(i,j) = fcn(i,j)
  - end do
  - end do
- Notice that only the outer loop variables are different from the serial solution.

© 2009 IBM Corporation

2-D Array Processing – an MPI Example ...

- Outline of the Solution with message passing:
- Implement as a Single Program Multiple Data (SPMD) model.
- Master process initializes array, sends info to worker processes and receives results.
- Worker process receives info, performs its share of computation and sends results to master.
- Using the Fortran storage scheme, perform block distribution of the array.

© 2009 IBM Corporation

2-D Array Processing – an MPI Example ...

- Pseudo code solution: red highlights changes for parallelism.
  - find out if I am MASTER or WORKER
  - if I am MASTER
  -   initialize the array
  -   send each WORKER info on part of array it owns
  -   send each WORKER its portion of initial array
  -   receive from each WORKER results
  - else if I am WORKER
  -   receive from MASTER info on part of array I own
  -   receive from MASTER my portion of initial array
  -   # calculate my portion of array
  -   do j = my first column,my last column
  -   do i = 1,n
  -     a(i,j) = fcn(i,j)
  -   end do
  -   end do
  -   send MASTER results
  - endif

© 2009 IBM Corporation

2-D Array Processing – an MPI Example ...

- Discussion on MPI program.

© 2009 IBM Corporation

Resources

© 2009 IBM Corporation

Reference

- This talk is heavily based around material provided in the following tutorial.
    - https://computing.llnl.gov/tutorials/parallel_comp/

© 2009 IBM Corporation