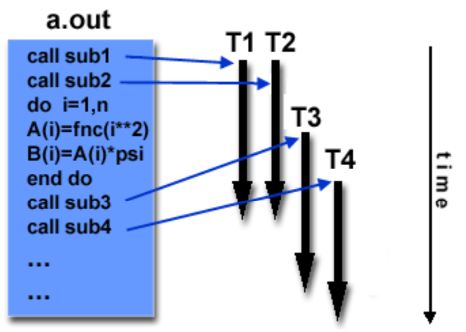
Overview of Programming Models

Parallel Programming Models

- There are several parallel programming models in common use:
  - Shared Memory
  - Threads
  - Message Passing
  - Data Parallel
  - Hybrid
- these models are NOT specific to a particular type of machine or memory architecture

Threads Model

- Unrelated standardization efforts have resulted in two very different implementations of threads: ***POSIX Threads*** and ***OpenMP***



POSIX vs OpenMP

POSIX Threads

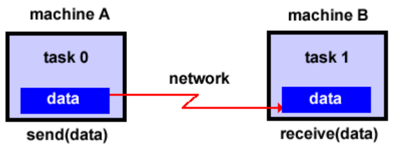
- Library based; requires parallel coding
- Specified by the IEEE POSIX 1003.1c standard (1995).
- C Language only
- Commonly referred to as Pthreads.
- Most hardware vendors now offer Pthreads in addition to their proprietary threads implementations.
- Very explicit parallelism; requires significant programmer attention to detail.

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"

Message Passing Model

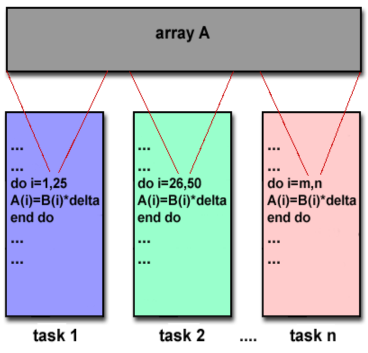
- A set of tasks that use their own local memory during computation. Multiple tasks can reside on the same physical machine as well across an arbitrary number of machines.
- Tasks exchange data through communications by sending and receiving messages.
- Data transfer usually requires cooperative operations to be performed by each process. For example, a send operation must have a matching receive operation.



www.mcs.anl.gov/Projects/mpi/standard.html.

Data Parallel Model

- Most of the parallel work focuses on performing operations on a data set. The data set is typically organized into a common structure, such as an array or cube.
- A set of tasks work collectively on the same data structure, however, each task works on a different partition of the same data structure.
- Tasks perform the same operation on their partition of work, for example, "add 4 to every array element".
- On shared memory architectures, all tasks may have access to the data structure through global memory. On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task.



Implementations

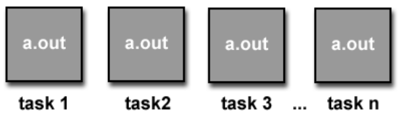
- Programming with the data parallel model is usually accomplished by writing a program with data parallel constructs. The constructs can be calls to a data parallel subroutine library or, compiler directives recognized by a data parallel compiler.
- **Compiler Directives:** Allow the programmer to specify the distribution and alignment of data. Fortran implementations are available for most common parallel platforms.
- Distributed memory implementations of this model usually have the compiler convert the program into standard code with calls to a message passing library (MPI usually) to distribute the data to all the processes. All message passing is done invisibly to the programmer.

Hybrid Models

- Combining message passing model (MPI) with either the threads model (POSIX threads) or the shared memory model (OpenMP).
  - This hybrid model lends itself well to the increasingly common hardware environment of networked SMP machines.
- Combining data parallel with message passing.
  - A data parallel implementations (F90, HPF) on distributed memory architectures actually use message passing to transmit data between tasks, transparently to the programmer.

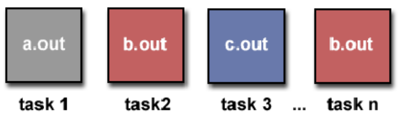
Single Program Multiple Data Model (SPMD)

- SPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- A single program is executed by all tasks simultaneously.
- At any moment in time, tasks can be executing the same or different instructions within the same program.
- SPMD programs usually have the necessary logic programmed into them to allow different tasks to branch or conditionally execute only those parts of the program they are designed to execute. That is, tasks do not necessarily have to execute the entire program - perhaps only a portion of it.
- All tasks may use different data



Multiple Program Multiple Data Model (MPMD)

- Like SPMD, MPMD is actually a "high level" programming model that can be built upon any combination of the previously mentioned parallel programming models.
- MPMD applications typically have multiple executable object files (programs). While the application is being run in parallel, each task can be executing the same or different program as other tasks.
- All tasks may use different data



Automatic VS Manual Parallelization

- A parallelizing compiler generally works in two different ways:
  - Fully Automatic
    - The compiler analyzes the source code and identifies opportunities for parallelism.
    - The analysis includes identifying inhibitors to parallelism and possibly a cost weighting on whether or not the parallelism would actually improve performance.
    - Loops (do, for) loops are the most frequent target for automatic parallelization.
  - Programmer Directed
    - Using "compiler directives" or possibly compiler flags, the programmer explicitly tells the compiler how to parallelize the code.
    - May be able to be used in conjunction with some degree of automatic parallelization also.

Caveats

- There are several important caveats that apply to automatic parallelization:
  - Wrong results may be produced
  - Performance may actually degrade
  - Much less flexible than manual parallelization
  - Limited to a subset (mostly loops) of code
  - May actually not parallelize code if the analysis suggests there are inhibitors or the code is too complex
  - Most automatic parallelization tools are for Fortran
- The remainder of this module applies to the manual method of developing parallel codes.

Understand the Problem

- 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 (1,1,2,3,5,8,13,21,...) by use of the formula: F(k + 2) = F(k + 1) + F(k)

  - This is a non-parallelizable problem because the calculation of the Fibonacci sequence as shown would entail dependent calculations rather than independent ones. The calculation of the k + 2 value uses those of both k + 1 and k. These three terms cannot be calculated independently and therefore, not in parallel.

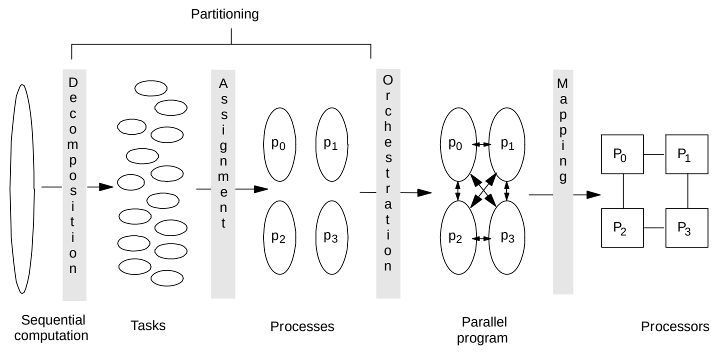
Designing Parallel Programs

- 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.
- 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
- Identify inhibitors to parallelism. One common class of inhibitor is *data dependence*, as demonstrated by the Fibonacci sequence above.
- Investigate other algorithms if possible. This may be the single most important consideration when designing a parallel application.

Definitions

- *Task*:
  - Arbitrary piece of work in parallel computation
  - Executed sequentially; concurrency is only across tasks
  - E.g. a particle/cell in Barnes-Hut, a ray or ray group in Raytrace
  - Fine-grained versus coarse-grained tasks
- *Process (thread)*:
  - Abstract entity that performs the tasks assigned to processes
  - Processes communicate and synchronize to perform their tasks
- *Processor*:
  - Physical engine on which process executes
  - Processes virtualize machine to programmer
    - write program in terms of processes, then map to processors

4 Steps in Creating a Parallel Program



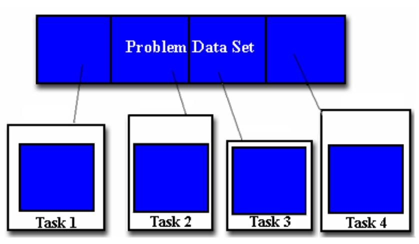
- Decomposition of computation in tasks
- Assignment of tasks to processes
- Orchestration of data access, comm, synch.
- Mapping processes to processors

Decomposition

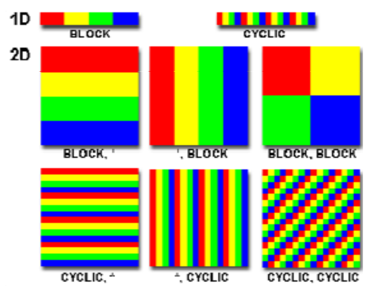
- Identify concurrency and decide level at which to exploit it
- Break up computation into tasks to be divided among processes
  - Tasks may become available dynamically
  - No. of available tasks may vary with time
- Goal: Enough tasks to keep processes busy, but not too many
  - Number of tasks available at a time is upper bound on achievable speedup
- There are two basic ways to partition computational work among parallel tasks: ***domain decomposition*** and ***functional decomposition***.

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

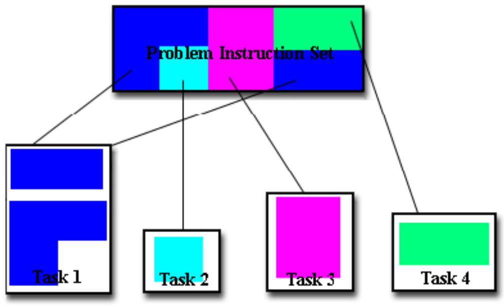


- There are different ways to partition data:



Functional Decomposition

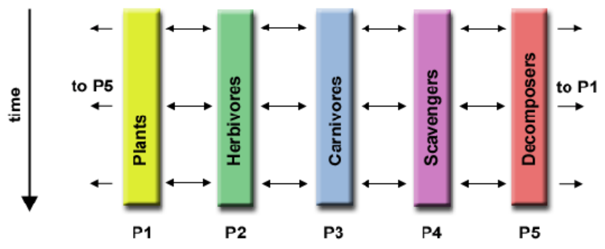
- In this approach, the focus is on the computation that is to be performed rather than on the data manipulated by the computation. The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work



- Functional decomposition lends itself well to problems that can be split into different tasks

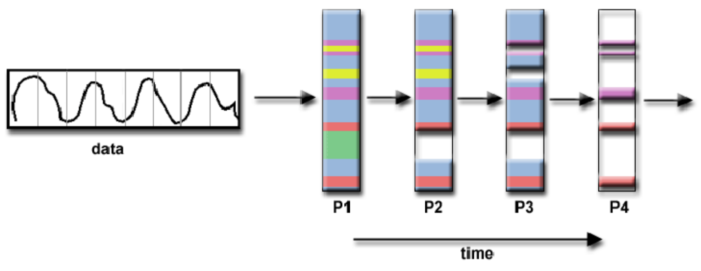
Ecosystem Modeling

- Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time step.



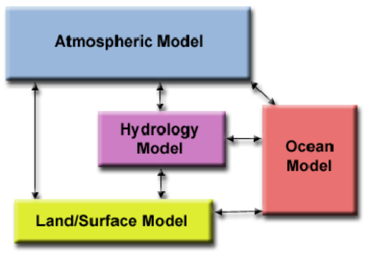
Signal Processing

- An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy.



Climate Modeling

- Each model component can be thought of as a separate task. Arrows represent exchanges of data between components during computation: the atmosphere model generates wind velocity data that are used by the ocean model, the ocean model generates sea surface temperature data that are used by the atmosphere model, and so on.



Communications

- Who Needs Communications? 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, imagine an image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work.
  - 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.

Factors to Consider

- **Cost of communications**
  - Inter-task communication virtually always implies overhead.
  - Machine cycles and resources that could be used for computation are instead used to package and transmit data.
  - Communications frequently require some type of synchronization between tasks, which can result in tasks spending time "waiting" instead of doing work.
  - Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.
- **Latency vs. Bandwidth**
  - ***latency*** is the time it takes to send a minimal (0 byte) message from point A to point B. Commonly expressed as microseconds.
  - ***bandwidth*** is the amount of data that can be communicated per unit of time. Commonly expressed as megabytes/sec.
  - Sending many small messages can cause latency to dominate communication overheads. Often it is more efficient to package small messages into a larger message, thus increasing the effective communications bandwidth.

Factors to Consider

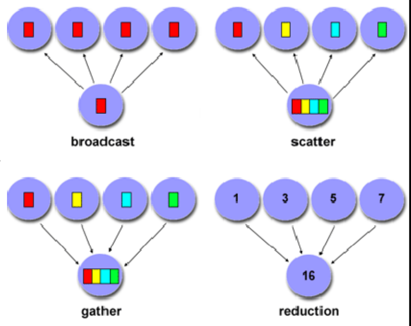
- **Visibility of communications**
  - With the Message Passing Model, communications are explicit and generally quite visible and under the control of the programmer.
  - With the Data Parallel Model, communications often occur transparently to the programmer, particularly on distributed memory architectures. The programmer may not even be able to know exactly how inter-task communications are being accomplished.

Factors to Consider

- **Synchronous vs. asynchronous communications**
  - Synchronous communications require some type of "handshaking" between tasks that are sharing data. This can be explicitly structured in code by the programmer, or it may happen at a lower level unknown to the programmer.
  - Synchronous communications are often referred to as ***blocking*** communications since other work must wait until the communications have completed.
  - Asynchronous communications allow tasks to transfer data independently from one another. For example, task 1 can prepare and send a message to task 2, and then immediately begin doing other work. When task 2 actually receives the data doesn't matter.
  - Asynchronous communications are often referred to as ***non-blocking*** communications since other work can be done while the communications are taking place.
  - Interleaving computation with communication is the single greatest benefit for using asynchronous communications.

Factors to Consider

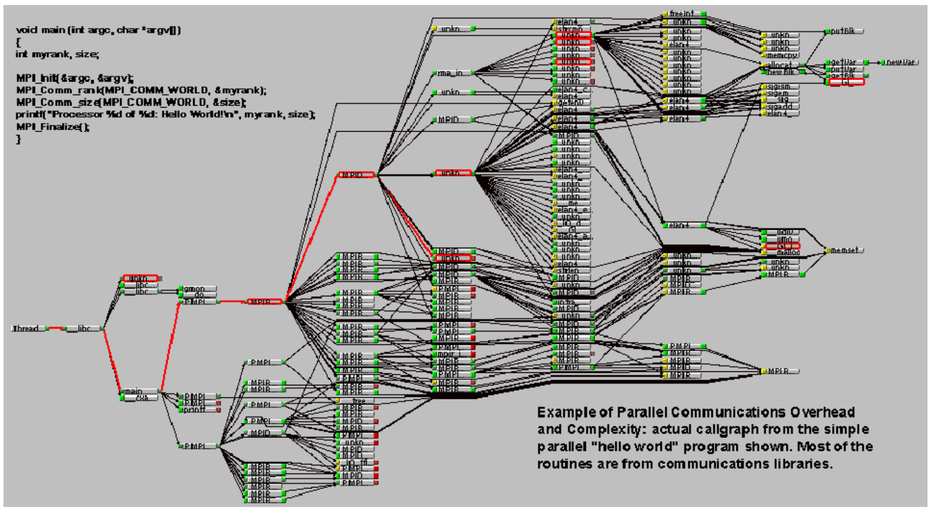
- **Scope of communications**
  - Knowing which tasks must communicate with each other is critical during the design stage of a parallel code. Both of the two scopings described below can be implemented synchronously or asynchronously.
  - ***Point-to-point*** - involves two tasks with one task acting as the sender/producer of data, and the other acting as the receiver/consumer.
  - ***Collective*** - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more):



Factors to Consider

- **Efficiency of communications**
  - Very often, the programmer will have a choice with regard to factors that can affect communications performance. Only a few are mentioned here.
  - Which implementation for a given model should be used? Using the Message Passing Model as an example, one MPI implementation may be faster on a given hardware platform than another.
  - What type of communication operations should be used? As mentioned previously, asynchronous communication operations can improve overall program performance.
  - Network media - some platforms may offer more than one network for communications. Which one is best?

Overhead and Complexity



```
void main (int argc, char *argv[])
{
int myrank, size;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
printf("Processor %d of %d: Hello World!\n", myrank, size);
MPI_Finalize();
}
```

Example of Parallel Communications Overhead and Complexity: actual callgraph from the simple parallel "hello world" program shown. Most of the routines are from communications libraries.

Synchronization

- **Barrier**
  - Usually implies that all tasks are involved
  - 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.
  - What happens from here varies. Often, a serial section of work must be done. In other cases, the tasks are automatically released to continue their work.
- **Lock / semaphore**
  - Can involve any number of tasks
  - 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.
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking

Synchronization

- **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.
  - Discussed previously in the Communications section.

Data Dependencies

- Definition:
  - A ***dependence*** exists between program statements when the order of statement execution affects the results of the program.
  - A ***data dependence*** results from multiple use of the same location(s) in storage by different tasks.
- Dependencies are important to parallel programming because they are one of the primary inhibitors to parallelism.

Examples

- **Loop carried data dependence**

```
DO 500 J = MYSTART,MYEND
A(J) = A(J-1) * 2.0
500 CONTINUE
```

- The value of A(J-1) must be computed before the value of A(J), therefore A(J) exhibits a data dependency on A(J-1). Parallelism is inhibited.
- If Task 2 has A(J) and task 1 has A(J-1), computing the correct value of A(J) necessitates:
  - Distributed memory architecture - task 2 must obtain the value of A(J-1) from task 1 after task 1 finishes its computation
  - Shared memory architecture - task 2 must read A(J-1) after task 1 updates it

Examples

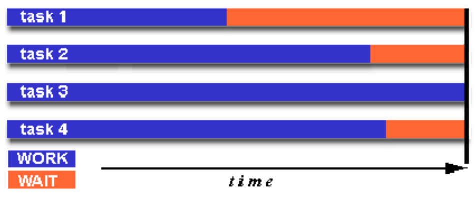
- **Loop independent data dependence**

```
task 1              task 2
------              ------
X = 2               X = 4
.                    .
.                    .
Y = X**2            Y = X**3
```

- As with the previous example, parallelism is inhibited. The value of Y is dependent on:
  - Distributed memory architecture - if or when the value of X is communicated between the tasks.
  - Shared memory architecture - which task last stores the value of X.
- Although all data dependencies are important to identify when designing parallel programs, loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts.
- How to Handle Data Dependencies:
  - Distributed memory architectures - communicate required data at synchronization points.
  - Shared memory architectures -synchronize read/write operations between tasks.

Load Balancing

- Load balancing refers to the practice of distributing work among tasks so that ***all*** tasks are kept busy ***all*** of the time. It can be considered a minimization of task idle time.
- Load balancing is important to parallel programs for performance reasons. For example, if all tasks are subject to a barrier synchronization point, the slowest task will determine the overall performance.



How to Achieve Load Balance

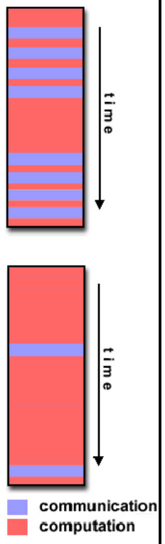
- **Equally partition the work each task receives**
  – For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  – For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  – If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

How to Achieve Load Balance

- **Use dynamic work assignment**
  - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
    - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
    - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
    - *N*-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.
  - When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a ***scheduler - task pool*** approach. As each task finishes its work, it queues to get a new piece of work.
  - It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Granularity

- Computation / Communication Ratio:
  - In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.
  - Periods of computation are typically separated from periods of communication by synchronization events.
- Fine-grain Parallelism:
  - Relatively small amounts of computational work are done between communication events
  - Low computation to communication ratio
  - Facilitates load balancing
  - Implies high communication overhead and less opportunity for performance enhancement
  - If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



Granularity

- Coarse-grain Parallelism:
  - Relatively large amounts of computational work are done between communication/synchronization events
  - High computation to communication ratio
  - Implies more opportunity for performance increase
  - Harder to load balance efficiently
- Which is Best?
  - The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
  - In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
  - Fine-grain parallelism can help reduce overheads due to load imbalance.

Limits and Costs

- Andahl's law states that potential program speedup is defined by the fraction of code (P) that can be parallelized:

  1
- **speedup = --------**

  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)
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

  1
- **speedup = ------------**

  P + S

  N
- where P = parallel fraction, N = number of processors and S = serial fraction

Limits of Parallelization

- It soon becomes obvious that there are limits to the scalability of parallelism. For example, at P = .50, .90 and .99 (50%, 90% and 99% of the code is parallelizable)

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1000	1.99	9.91	90.99
10000	1.99	9.91	99.02

Scalable Problems

- However, certain problems demonstrate increased performance by increasing the problem size. For example:
  - **2D Grid Calculations 85 seconds 85%**
  - **Serial fraction 15 seconds 15%**
- We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:
  - **2D Grid Calculations 680 seconds 97.84%**
  - **Serial fraction 15 seconds 2.16%**
- Problems that increase the percentage of parallel time with their size are more ***scalable*** than problems with a fixed percentage of parallel time.

Examples

- Array processing
- PI calculation
- Heat equation
- 1-D wave equation