Designing Parallel Programs

Automatic vs. Manual Parallelization

- Designing and developing parallel programs has characteristically been a very manual process. The programmer is typically responsible for both identifying and actually implementing parallelism.
- Very often, manually developing parallel codes is a time consuming, complex, error-prone and ***iterative*** process.
- For a number of years now, various tools have been available to assist the programmer with converting serial programs into parallel programs. The most common type of tool used to automatically parallelize a serial program is a parallelizing compiler or pre-processor.

Automatic vs. Manual Parallelization (2)

- 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) are the most frequent target for automatic parallelization.
    - Example: Intel compilers (default in SHARCNET), with the "`-parallel`" switch.
  - 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.
    - Intel compilers: C pragma (`#pragma parallel`) or Fortran directive (`!DIR$ PARALLEL`)

Automatic vs. Manual Parallelization (3)

- If you are beginning with an existing serial code and have time or budget constraints, then automatic parallelization may be the answer. However, 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
- The remainder of this section applies to the manual method of developing parallel codes.

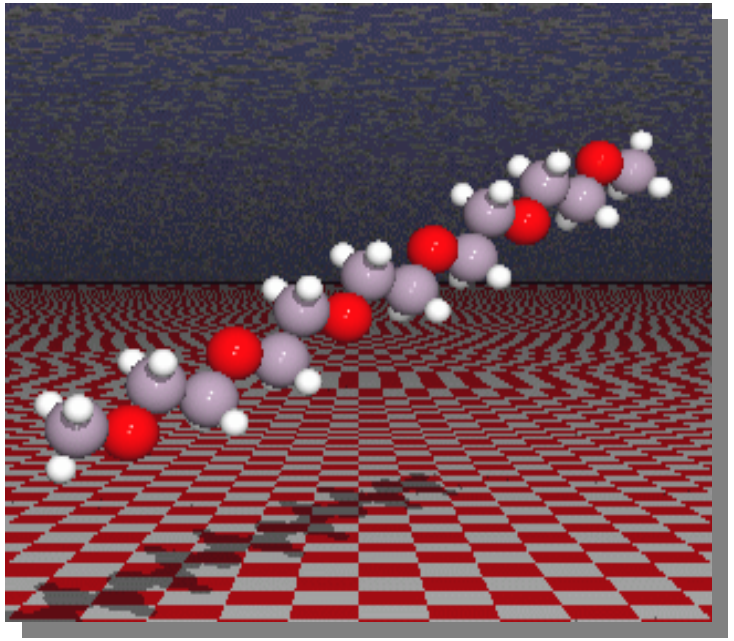
Understand the Problem and the Program

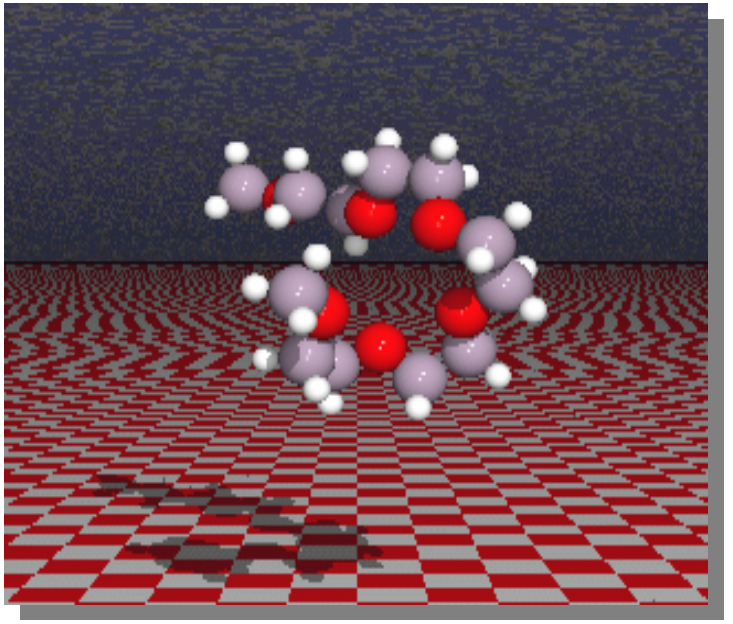
- Undoubtedly, the first step in developing parallel software is to first understand the problem that you wish to solve in parallel. If you are starting with a serial program, this necessitates understanding the existing code also.
- Before spending time in an attempt to develop a parallel solution for a 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 can be solved in parallel. Each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation is also a parallelizable problem.

Molecule conformations





Understand the Problem and the Program (2)

– 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.

Understand the Problem and the Program (3)

- 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.

Understand the Problem and the Program (4)

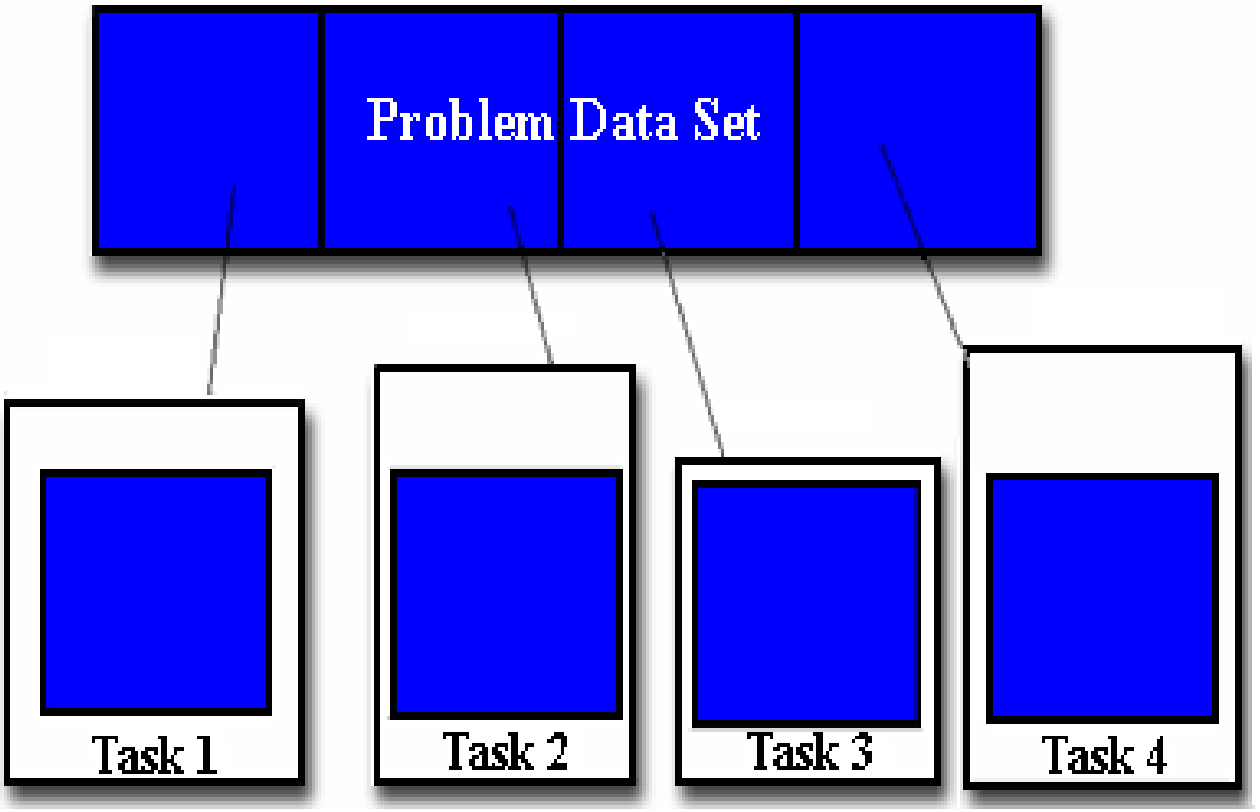
- Identify bottlenecks in the program
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? For example, I/O is usually something that slows a program down.
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate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

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** and **functional decomposition**.

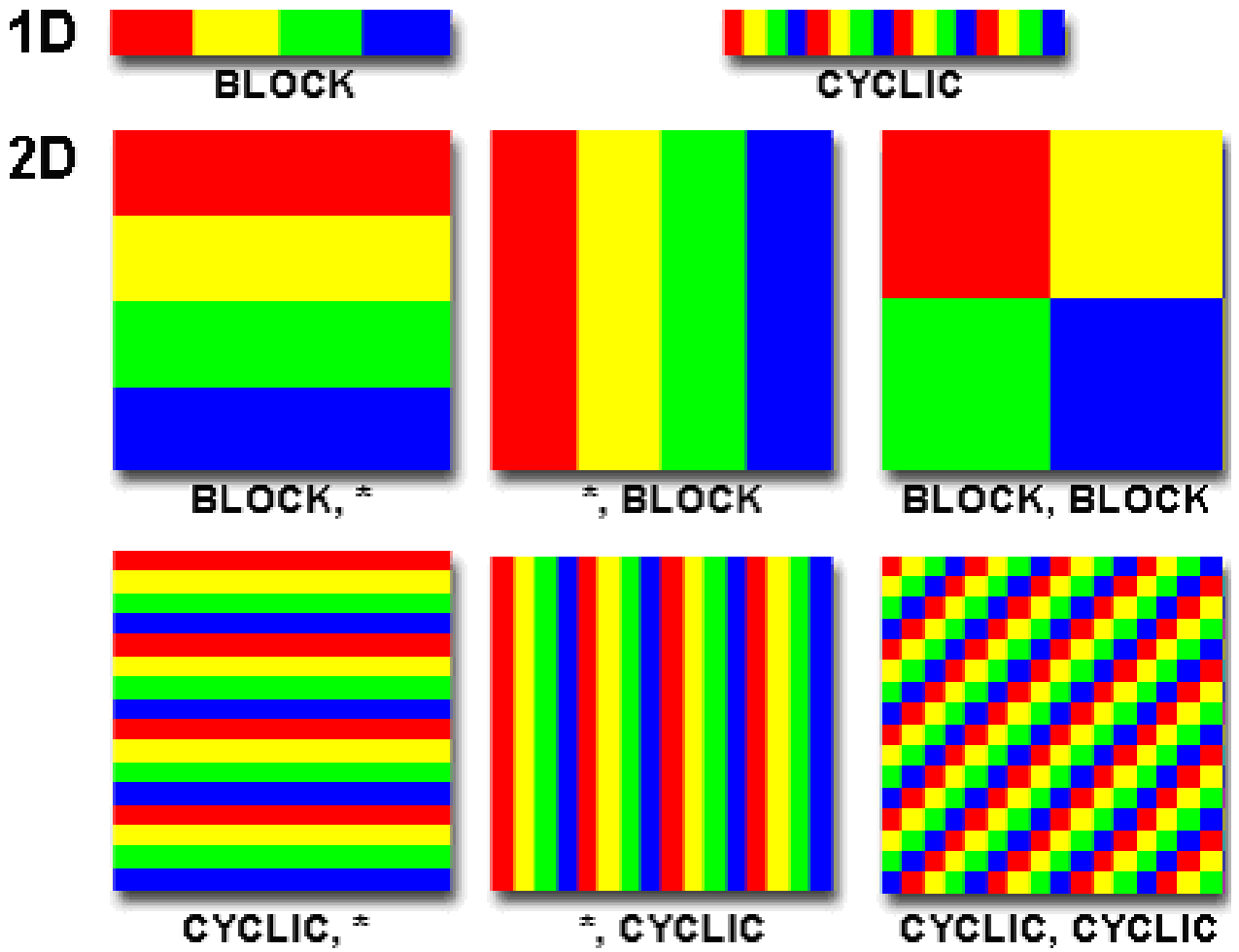
Domain Decomposition

- 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.



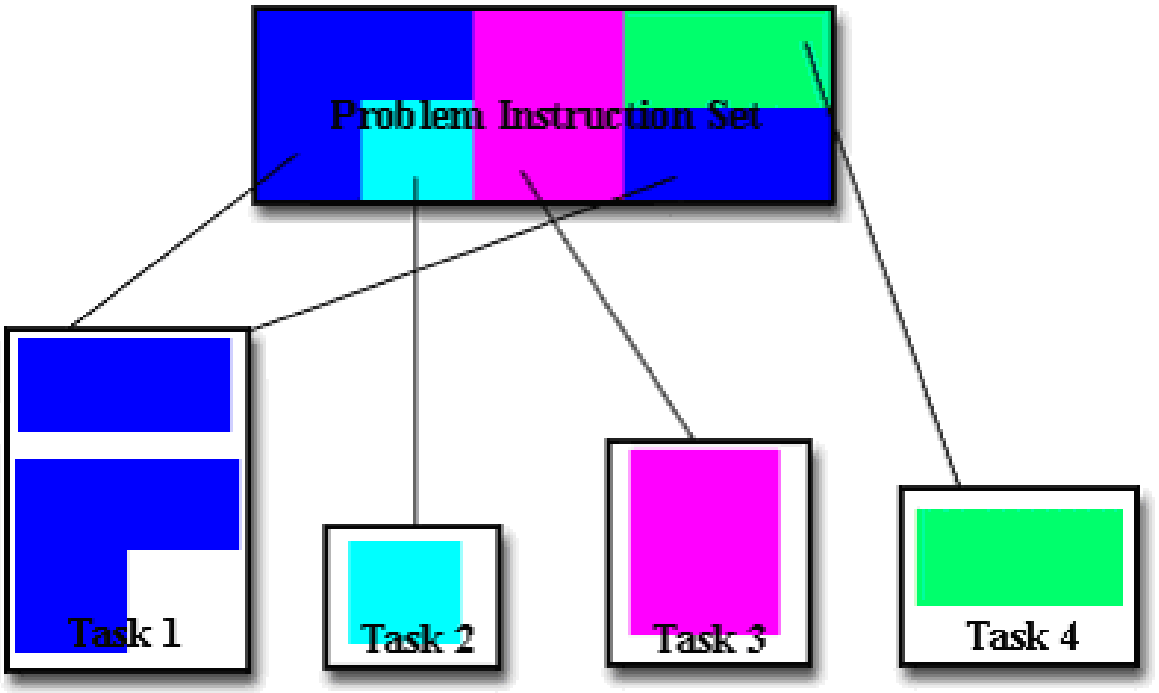
Domain Decomposition (2)

- 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.

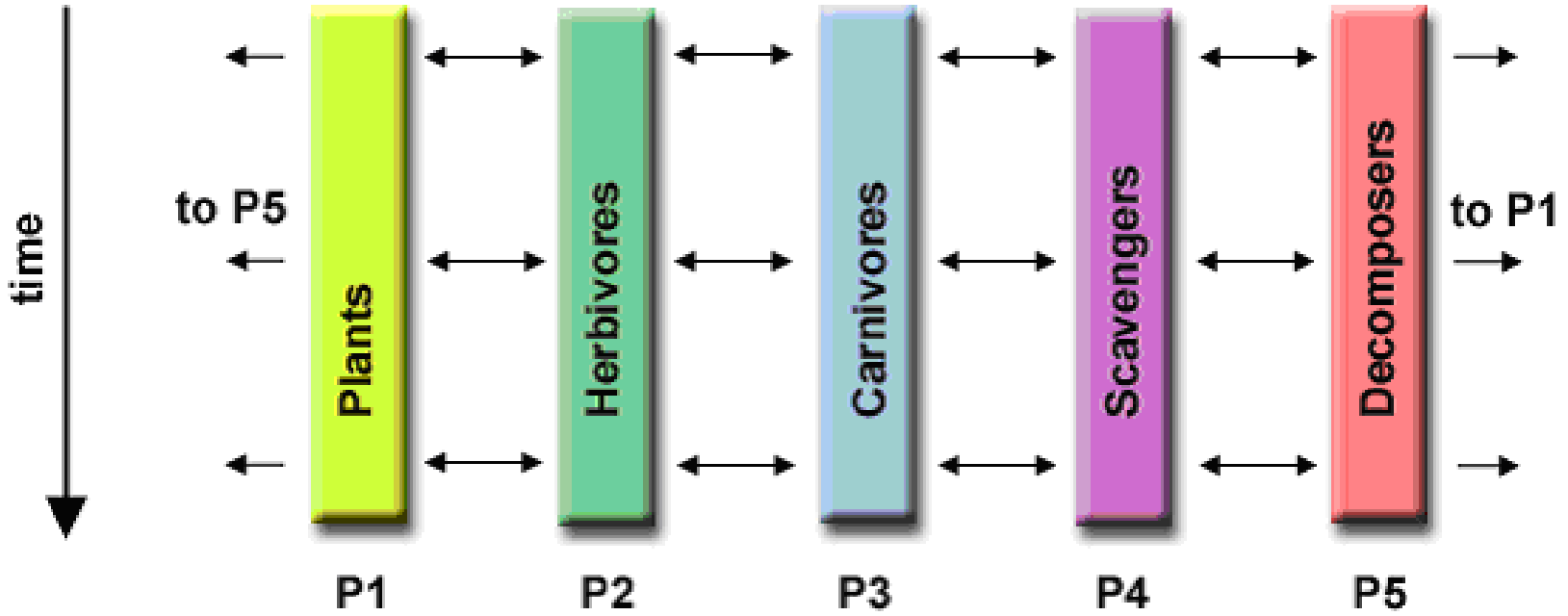


Functional Decomposition (2)

- Functional decomposition lends itself well to problems that can be split into different tasks. For example:

Ecosystem Modeling

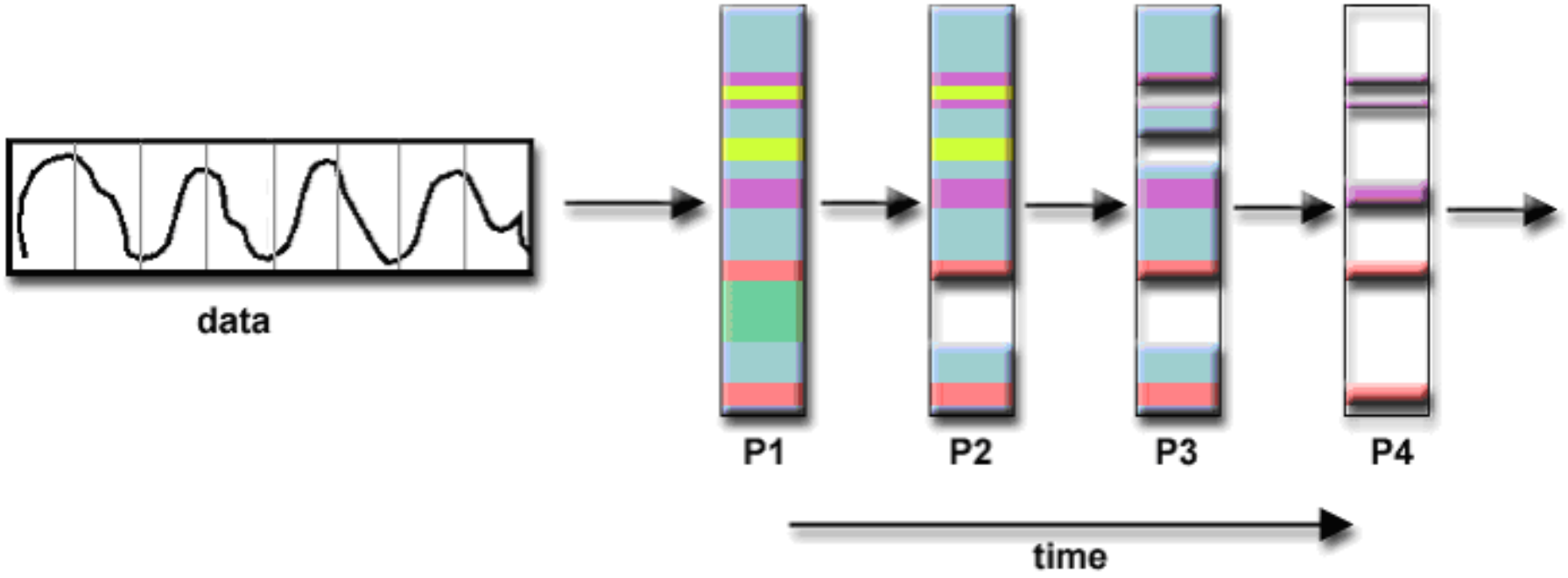
Each program calculates the population of a given group, where each group's growth depends on that of its neighbors. As time progresses, each process calculates its current state, then exchanges information with the neighbor populations. All tasks then progress to calculate the state at the next time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



Functional Decomposition (3)

Signal Processing

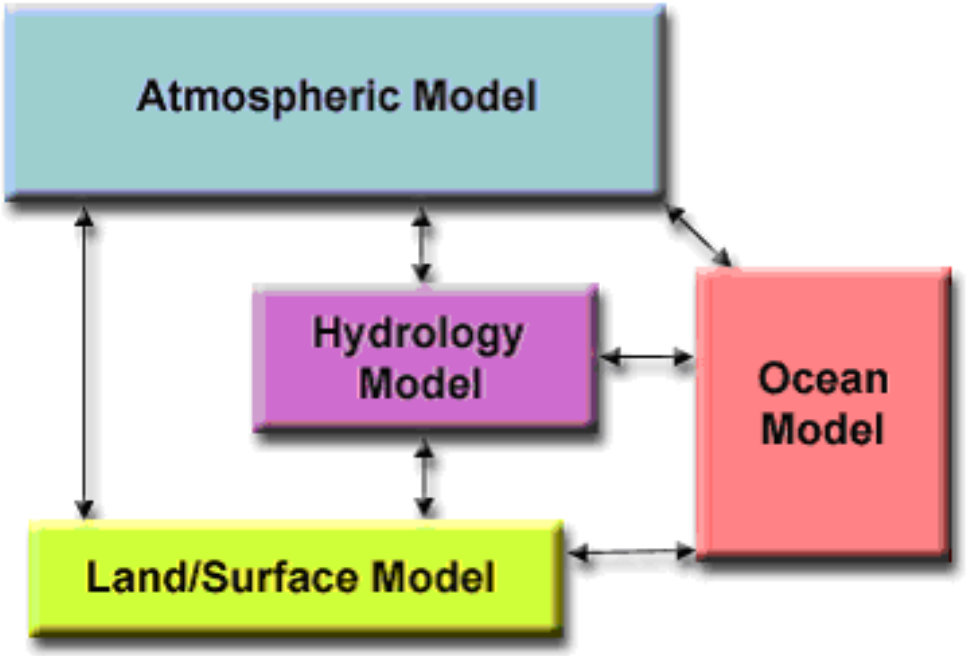
An audio signal data set is passed through four distinct computational filters. Each filter is a separate process. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth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



Functional Decomposition (4)

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.



- Combining these two types of problem decomposition is common and natural.

Designing Parallel Programs:

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 (2)

- **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.

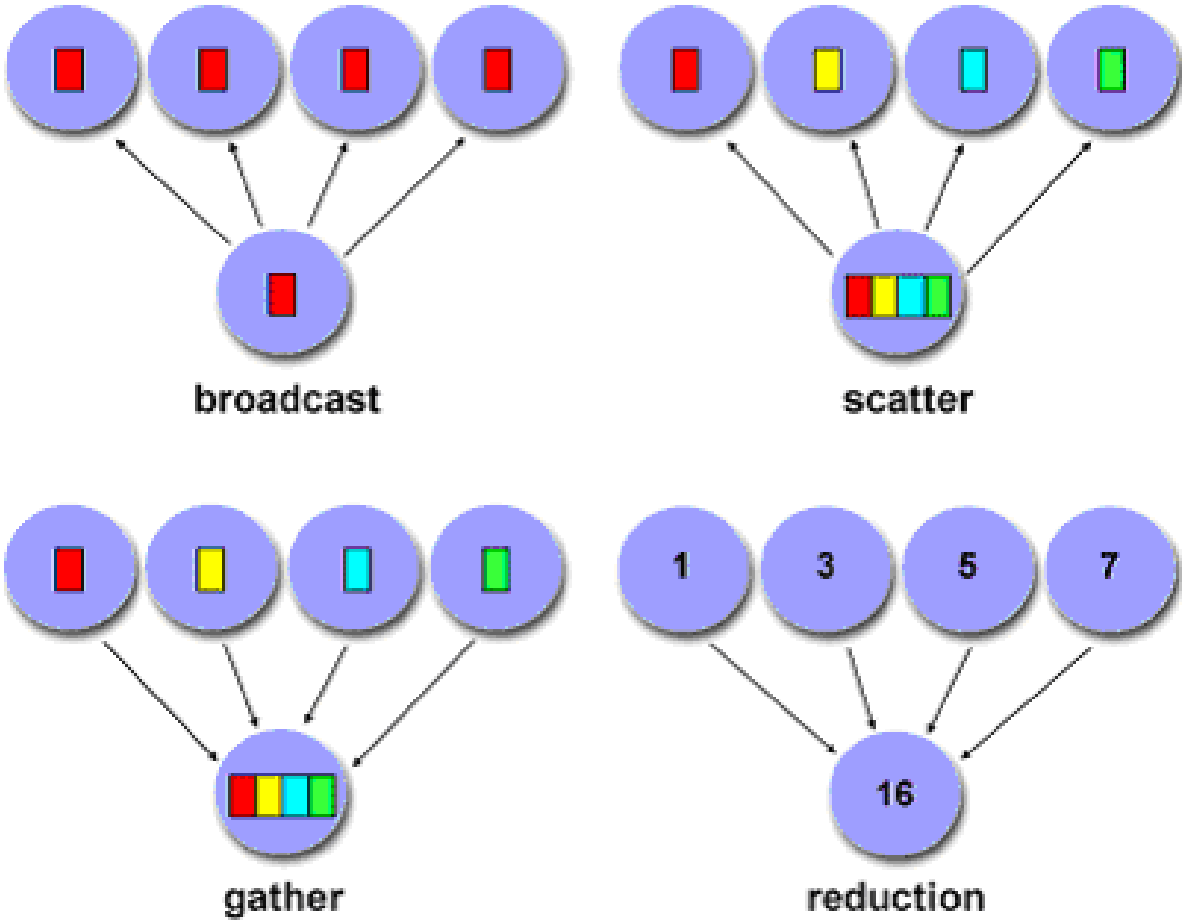
Factors to Consider (3)

- **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 (4)

- **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 (5)



Factors to Consider (6)

- **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?

Designing Parallel Programs:

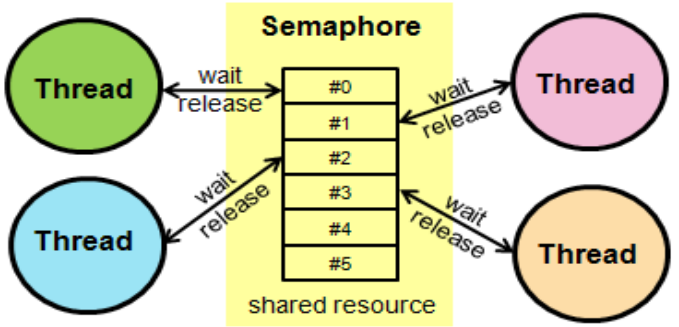
Synchronization

Types of 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.

Types of Synchronization (2)

- 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



- **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.

Designing Parallel Programs:

Data Dependencies

Loop carried dependency

- Dependency between statements executed in different iterations of the loop.
- Dependencies are always associated with a particular memory location, we can detect them by analyzing how each variable is used within the loop
  - Is the variable only read and never assigned within the loop body? If so, there are no dependencies involving it
  - Otherwise, consider the memory locations that make up the variable and that are assigned within the loop. For each such location, is there exactly one iteration that accesses the location? If so, there are no dependencies involving the variable. If not, there is a dependency.

Loops with or without data dependency

1
```
do i = 2, n
    a(i) = a(i) + a(i-1)
enddo
```

2
```
do i = 2, n, 2
    a(i) = a(i) + a(i-1)
enddo
```

3
```
do i = 1, n/2
    a(i) = a(i) + a(i + n/2)
enddo
```

4
```
do i = 1, n/2+1
    a(i) = a(i) + a(i + n/2)
enddo
```

1 yes

each iteration writes an element of a that is read by the next iteration

2 no

loop has a stride of 2, it writes every other element

3 no

each iteration reads only the element it writes to plus an element that is not written to by the loop since it has a subscript greater than n/2

4 yes

the first iteration reads a(n/2+1), while that last iteration writes to this element

Classification

- Data-flow dependency

Data-flow relation between two dependent statements, i.e., whether or not the two statements communicate values through the memory location.

S1 – earlier statement, writes to a memory location
S2 – later statement, reads the memory location

The value read by S2 in a serial execution is the same as that written by S1. In this case, the result of a computation by S1 is communicated, or 'flows' to S2, hence called "flow dependency".

S1 must execute first to produce the value that is read by S2.

Generally, it's hard to remove this dependency.

Classification: continue

Two other kinds of dependencies which can be removed; they are not communication of data between S1 and S2, but reuse of the memory for different purposes at different points in the program.

- Anti dependency

S1 reads the location
S2 writes to the location

Solution: make a private copy of the location and use it to initialize S1.

- Output dependency

Both S1 and S2 write to one location

Solution: make the memory location private (to a thread) and in addition copy the last value back to the shared copy of the location.

A loop containing multiple data dependencies

```
    do i = 2, n-1
10      x = d(i) + i
20     a(i) = a(i + 1) + x
30     b(i) = b(i) +b(i - 1) + d(i - 1)
40     c(2) = 2 * i
    enddo
```

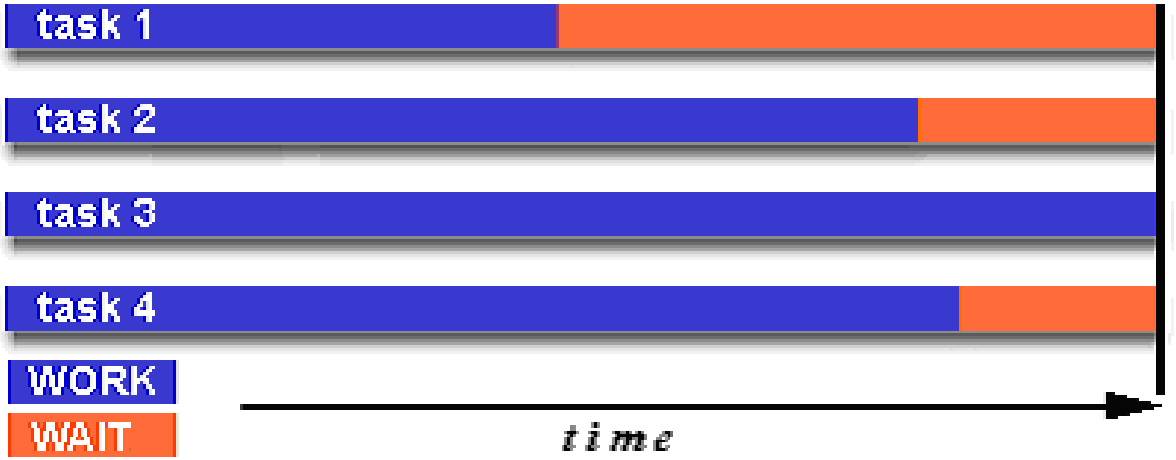
Memory location	Line	Iteration	Access	Line	Iteration	Access	Loop carried?	Kind of dependency
x	10	i	w	20	i	r	n	flow
x	10	i	w	10	i+1	w	y	output
x	20	i	r	10	i+1	w	y	anti
a(i+1)	20	i	r	20	i+1	w	y	anti
b(i)	30	i	w	30	i+1	r	y	flow
c(2)	40	i	w	40	i+1	w	y	output

Designing Parallel Programs:

Load Balancing

Definition

- 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

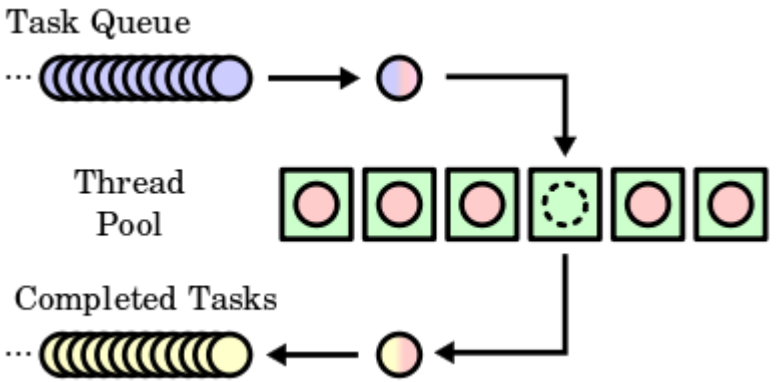
- **Equally partition the work each task receives**
  - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - If 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 (2)

- **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.

How to Achieve Load Balance (3)

- 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.



Designing Parallel Programs:

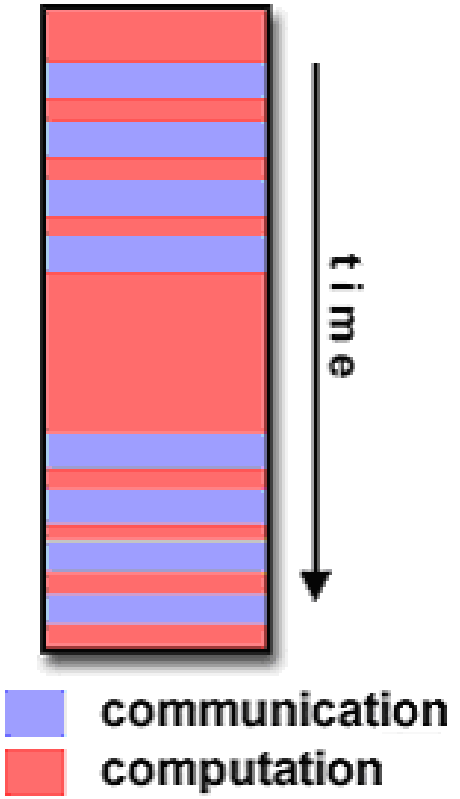
Granularity

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.

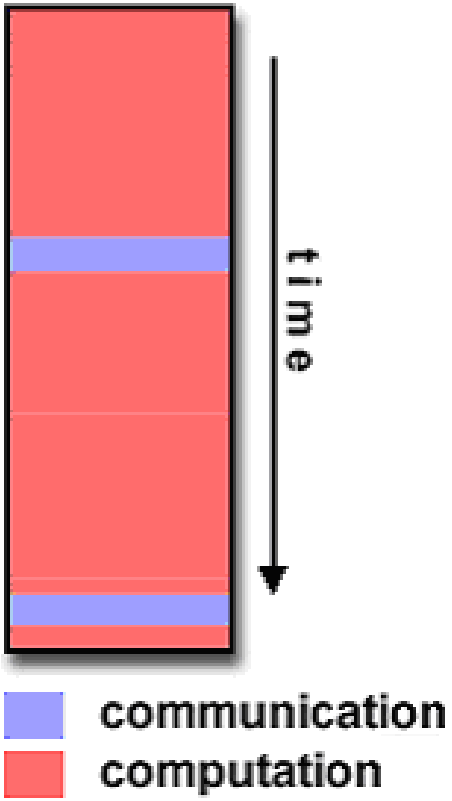
Fine-grain Parallelism

- Relatively small amounts of computational work are done between communication events
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- Low computation to communication ratio (because of the high communication overhead)
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



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.

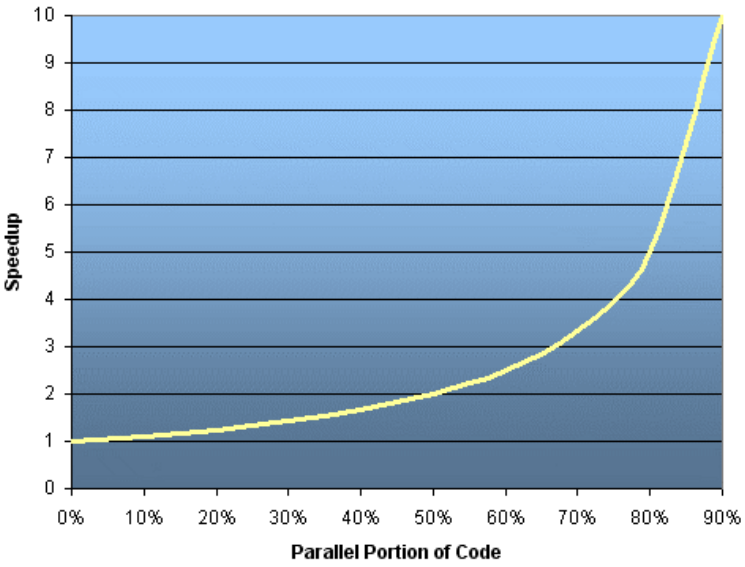
Designing Parallel Programs:

Limits and Costs 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:

$$\text{speedup} = \frac{1}{1 - P}$$



- 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.

Amdahl's Law (2)

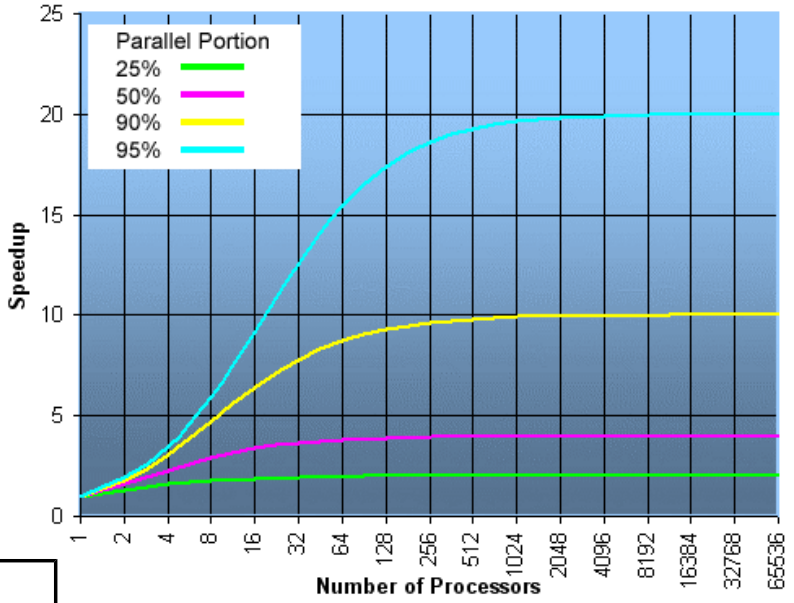
- Introducing the number of processors performing the parallel fraction of work, the relationship can be modeled by:

$$\text{speedup} = \frac{1}{\frac{P}{N} + S}$$

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

Amdahl's Law (3)

- 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

Gustafson's law

- *Gustafson's Law* is a law in computer engineering which states that any sufficiently large problem can be efficiently parallelized.
- Gustafson's Law is closely related to Amdahl's law, which gives a limit to the degree to which a program can be sped up due to parallelization. It was first described by John Gustafson in 1988.

$$Speedup(N) = N - S\ (N - 1).$$

N... number of processors, S ... non-parallelizable part of process

- Gustafson's law addresses the shortcomings of Amdahl's law which cannot scale to match availability of computing power as the machine size increases.

Gustafson's law (2)

- It removes the fixed problem size or fixed computation load on the parallel processors, instead he proposed a fixed time concept which leads to scaled speed up.
- Amdahl's law is based on fixed workload or fixed problem size. It implies that the sequential part of a program does not change with respect to machine size (i.e, the number of processors). However the parallel part is evenly distributed by *N* processors.
- The impact of the law was the shift in research to develop parallelizing compilers and reduction in the serial part of the solution to boost the performance of parallel systems.
- Both Amdahl's and Gustafson's laws assume that communication overheads are negligible. In real systems, bandwidth limits and non-zero latency will prevent efficient scaling of most numerical codes to arbitrarily large numbers of CPUs.

Strong and Weak Scaling

- Strong Scaling: describes how does the time to solution vary with the number of processors for a fixed system size.

- Weak Scaling: is how the time to solution varies with processor count with a fixed system size per processor.

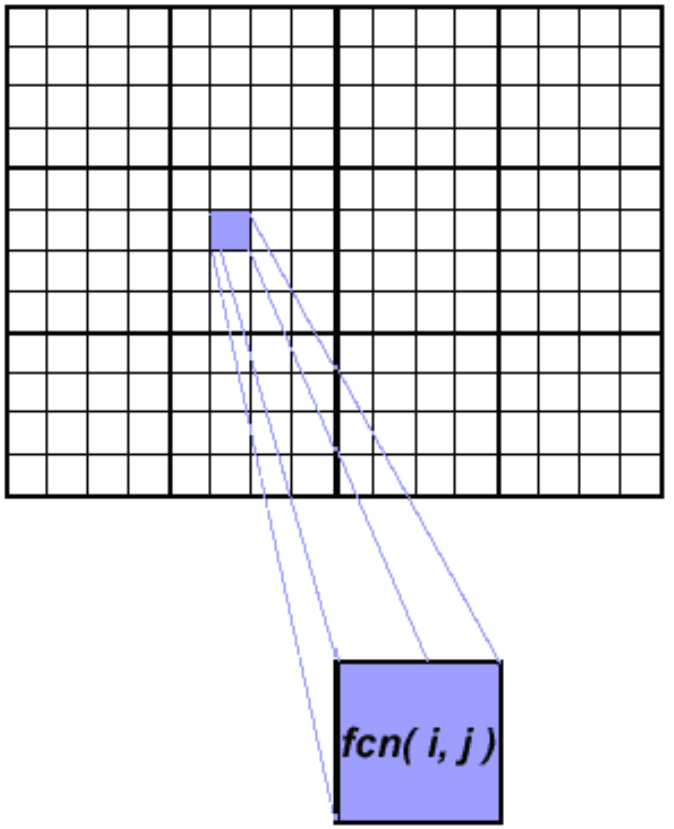
Parallel Examples

Array Processing

Setup

- 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:

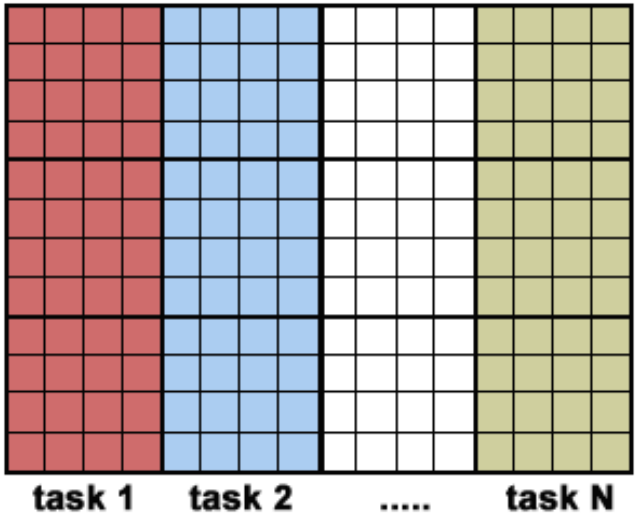
```
do j = 1,n
  do i = 1,n
    a(i,j) = fcn(i,j)
    end do
  end do
```



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

Solution 1

- Arrays elements are distributed so that each processor owns a portion of an array (subarray).
- Independent calculation of array elements insures 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. See the Block - Cyclic Distributions Diagram for the options.



Solution 1 (2)

- 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.

Solution 1 (3)

- **One Possible Solution:**
  - Implement as 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.
  - Pseudo code solution: **red** highlights changes for parallelism.

Solution 1 (4)

```
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
```

Solution 2: Pool of Tasks

- The previous array solution demonstrated static load balancing:
  - Each task has a fixed amount of work to do.
  - May be significant idle time for faster or more lightly loaded processors - slowest task determines overall performance.
- Static load balancing is not usually a major concern if all tasks are performing the same amount of work on identical machines.
- If you have a load balance problem (some tasks work faster than others), you may benefit by using a "pool of tasks" scheme.

Solution 2: Pool of Tasks (2)

Pool of Tasks Scheme:

- Two kinds of processes are employed

  Master Process:
  - Holds pool of tasks for worker processes to do
  - Sends worker a task when requested
  - Collects results from workers

  Worker Process: repeatedly does the following
  - Gets task from master process
  - Performs computation
  - Sends results to master

- Worker processes do not know before runtime which portion of array they will handle or how many tasks they will perform.
- Dynamic load balancing occurs at run time: the faster tasks will get more work to do.
- Pseudo code solution: **red** highlights changes for parallelism.

Solution 2: Pool of Tasks (3)

```
find out if I am MASTER or WORKER

    if I am MASTER

        do until no more jobs
            receive prev. results from WORKER
            send to WORKER next job
        end do

        tell WORKER no more jobs

    else if I am WORKER

        do until no more jobs
            send prev. results to MASTER
            receive from MASTER next job

            calculate array element: a(i,j)=fcn(i,j)


        end do

    endif
```

Solution 2: Pool of Tasks (4)

Discussion:

- In the above pool of tasks example, each task calculated an individual array element as a job. The computation to communication ratio is finely granular.
- Finely granular solutions incur more communication overhead in order to reduce task idle time.
- A more optimal solution might be to distribute more work with each job. The "right" amount of work is problem dependent.

Parallel Examples

Simple Heat Equation

Setup

- Most problems in parallel computing require communication among the tasks. A number of common problems require communication with "neighbor" tasks.
- The heat equation describes the temperature change over time, given initial temperature distribution and boundary conditions.
- A finite differencing scheme is employed to solve the heat equation numerically on a square region.
- The initial temperature is zero on the boundaries and high in the middle.
- The boundary temperature is held at zero.
- For the fully explicit problem, a time stepping algorithm is used. The elements of a 2-dimensional array represent the temperature at points on the square.

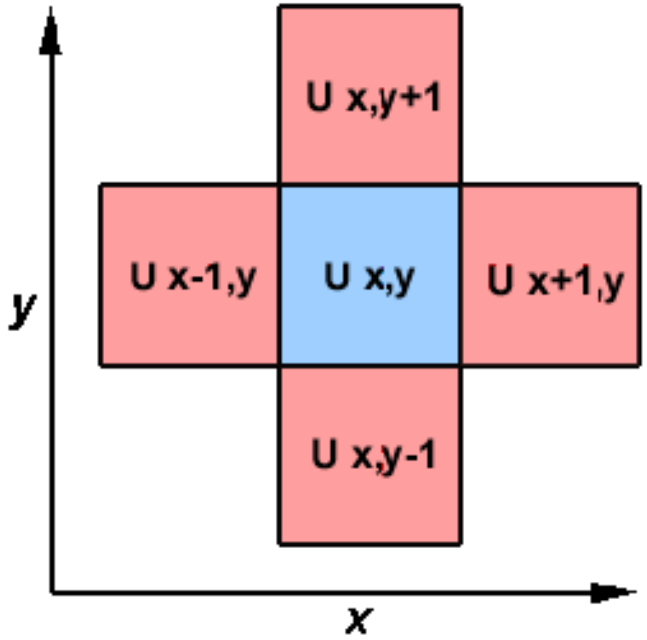
Setup (2)

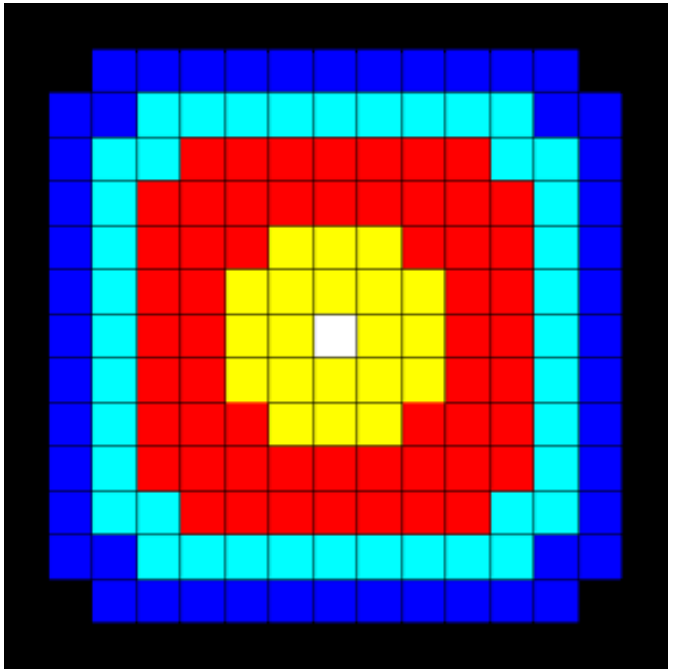
- The calculation of an element is dependent upon neighbor element values.

$$U_{x,y} = U_{x,y}$$

$$+ C_x * (U_{x+1,y} + U_{x-1,y} - 2 * U_{xy})$$

$$+ C_y * (U_{x,y+1} + U_{x,y-1} - 2 * U_{x,y})$$





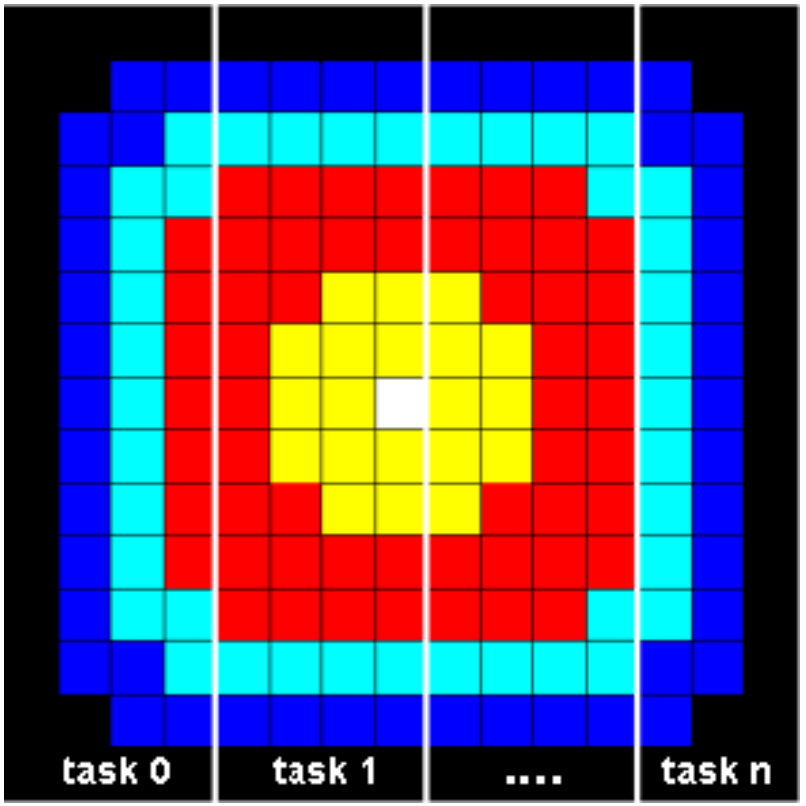
Setup (3)

- A serial program would contain code like:

```
do iy = 2, ny - 1
do ix = 2, nx - 1
      u2(ix, iy) =
          u1(ix, iy)  +
              cx  (u1(ix+1,iy) + u1(ix-1,iy) - 2.u1(ix,iy)) +
              cy  (u1(ix,iy+1) + u1(ix,iy-1) - 2.u1(ix,iy))
end do
end do
```

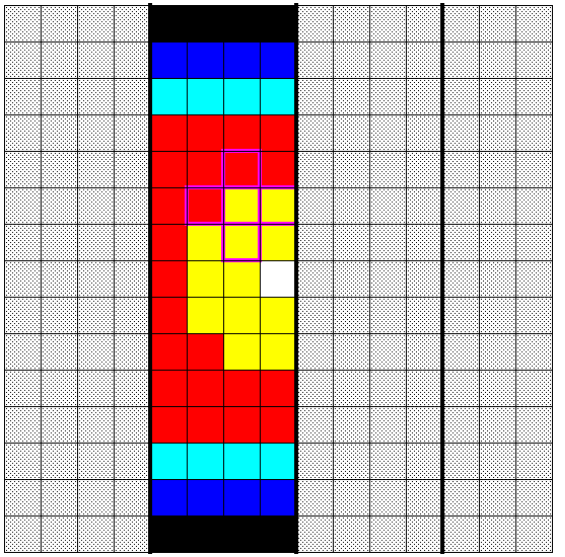
Parallel Solution 1

- Implement as an SPMD model
- The entire array is partitioned and distributed as subarrays to all tasks. Each task owns a portion of the total array.

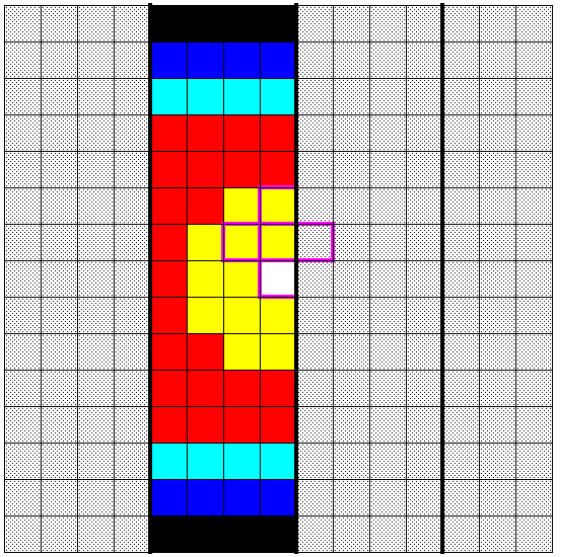


Parallel Solution 1 (2)

- Determine data dependencies
  - interior elements belonging to a task are independent of other tasks



  - border elements are dependent upon a neighbor task's data, necessitating communication.



Parallel Solution 1 (3)

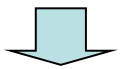
- Master process sends initial info to workers, checks for convergence and collects results
- Worker process calculates solution, communicating as necessary with neighbor processes
- Pseudo code solution: **red** highlights changes for parallelism.

```
find out if I am MASTER or WORKER

if I am MASTER
      initialize array
      send each WORKER starting info and subarray

      do until all WORKERS converge
           gather from all WORKERS convergence data
           broadcast to all WORKERS convergence signal
      end do

      receive results from each WORKER
```

Parallel Solution 1 (4)

```
else if I am WORKER
      receive from MASTER starting info and subarray

      do until solution converged
          update time
          send neighbors my border info
          receive from neighbors their border info

          update my portion of solution array

          determine if my solution has converged
               send MASTER convergence data
               receive from MASTER convergence signal
      end do

      send MASTER results

endif
```

Solution 2

- In the previous solution, it was assumed that blocking communications were used by the worker tasks. Blocking communications wait for the communication process to complete before continuing to the next program instruction.
- In the previous solution, neighbor tasks communicated border data, then each process updated its portion of the array.
- Computing times can often be reduced by using non-blocking (asynchronous) communication. Non-blocking communications allow work to be performed while communication is in progress.
- Each task could update the interior of its part of the solution array while the communication of border data is occurring, and update its border after communication has completed.
- Pseudo code for the second solution: **red** highlights changes for non-blocking communications.

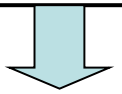
Solution 2 (2)

```
find out if I am MASTER or WORKER

if I am MASTER
      initialize array
      send each WORKER starting info and subarray

      do until all WORKERS converge
          gather from all WORKERS convergence data
          broadcast to all WORKERS convergence signal
      end do

      receive results from each WORKER
```



Solution 2 (3)

```
else if I am WORKER
      receive from MASTER starting info and subarray

      do until solution converged
          update time

          non-blocking send neighbors my border info
          non-blocking receive neighbors border info

          update interior of my portion of solution array
          wait for non-blocking communication complete
          update border of my portion of solution array

           determine if my solution has converged
               send MASTER convergence data
               receive from MASTER convergence signal
      end do

      send MASTER results

endif
```