1: Parallel Programming Concepts

John Burkardt
Information Technology Department
Virginia Tech

.....

HPPC-2008
High Performance Parallel Computing Bootcamp .....
http://people.sc.fsu.edu/~jburkardt/presentations/
hppc_2008_lecture1.pdf

28 July - 02 August
2008

Parallel Programming Concepts

Parallel Programming Concepts

The difference between 1,000 workers working on 1,000 projects, and 1,000 workers working on 1 project is **organization** and **communication**.

The key idea of parallel programming:

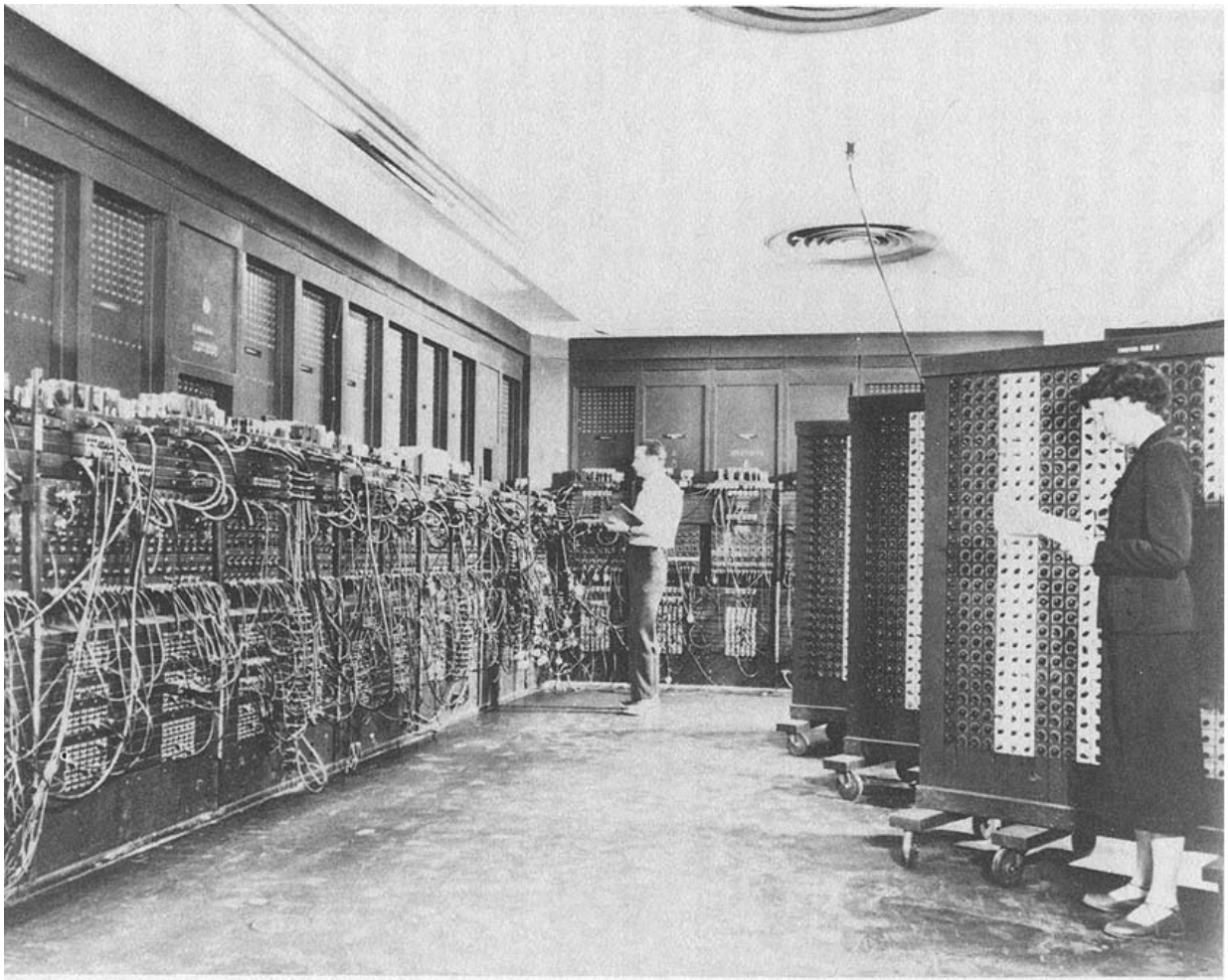
> *Independent agents, properly organized and able to communicate, can cooperate on one task.*

Parallel Programming Concepts

1. **Sequential Computing and its Limits**

Your next computer won't run any faster that the one you have.

Sequential Computing and its Limits



The ENIAC.
Smithsonian Institution Photo No. 53192.

ENIAC Weighed 30 Tons

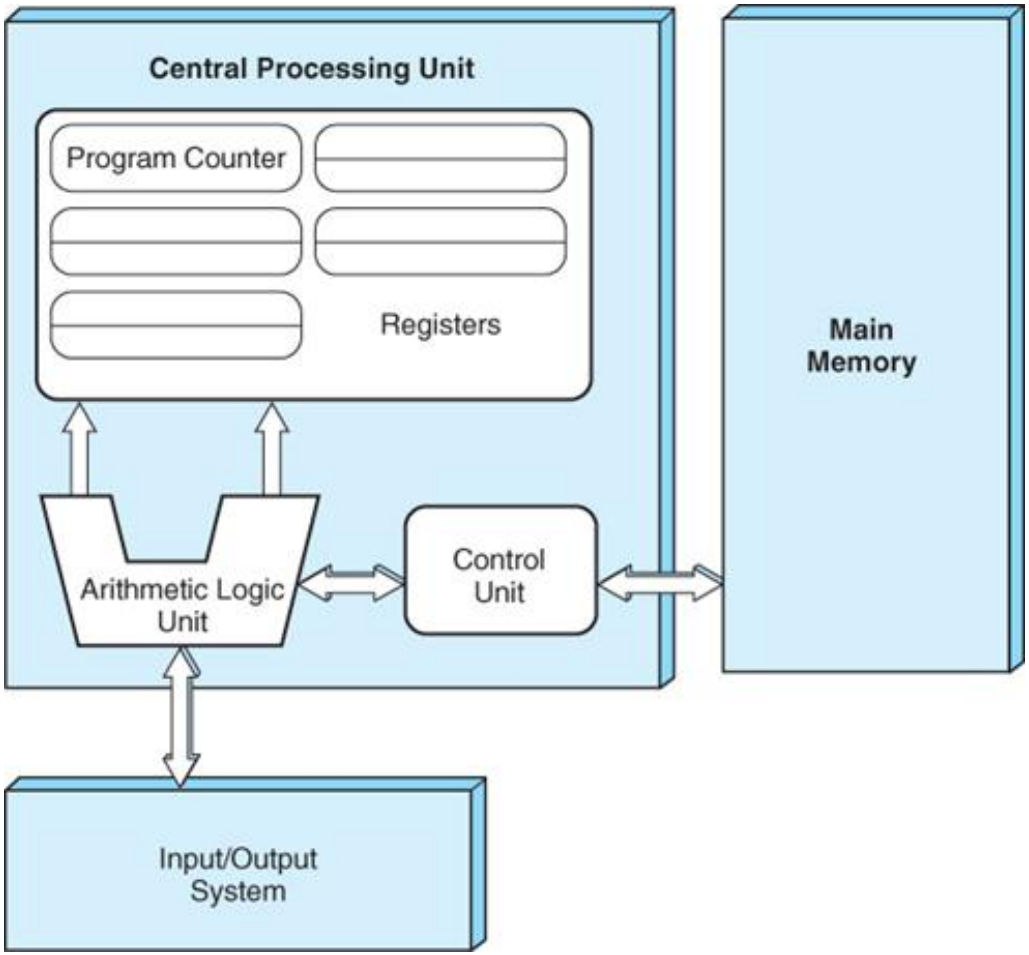
Sequential Computing and its Limits

John von Neumann's name appears on the cover of the user manual for the ENIAC computer, the first working electronic reprogrammable general purpose calculating device.

ENIAC was huge, heavy, and slow. Data was stored as voltage levels. To add two numbers required turning dials and plugging a connecting wire between the devices that stored the numbers.

Nonetheless, von Neumann's mental image of the logical structure of ENIAC guided computer designers for 50 years.

Sequential Computing and its Limits



The von Neumann Architecture

Sequential Computing and its Limits

The interesting part of the von Neumann architecture is, of course, the central processing unit or **CPU**:

- The **control unit** is in charge. It fetches a portion of the program from memory, it gets data from memory to "feed" the arithmetic unit, and it sends results back to memory.
- The **arithmetic-logic unit** is the raw computational device that carries out additions and multiplications and logical operations.
- The **registers** are a small working area of temporary data used during computations.

Sequential Computing and its Limits

Over time, faster electronics were put in the CPU but often the computations did not speed up as expected. It turned out that the CPU was *"starving"*, that is, it could compute results so fast that it was almost always idle, waiting for more data from memory.

For this reason, more connections to memory were added, but most importantly, a small fast **cache unit** was added to the CPU. The cache kept a local copy of certain data that was frequently used.

The cache was so useful that there are now elaborate multi-level caches, and sophisticated algorithms for guessing what data should be kept in the cache.

Sequential Computing and its Limits

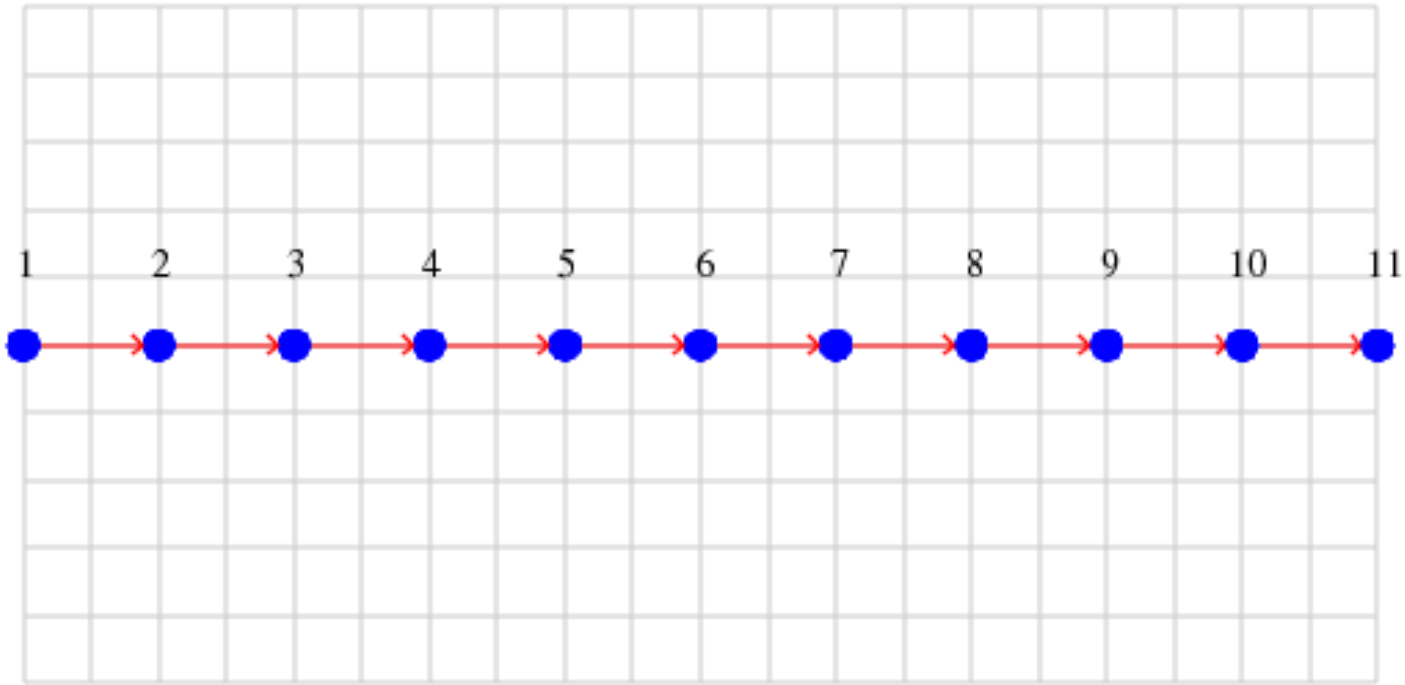
A sequential program carried out a computation

solve this system of linear equations

by breaking it down into a series of simple steps:

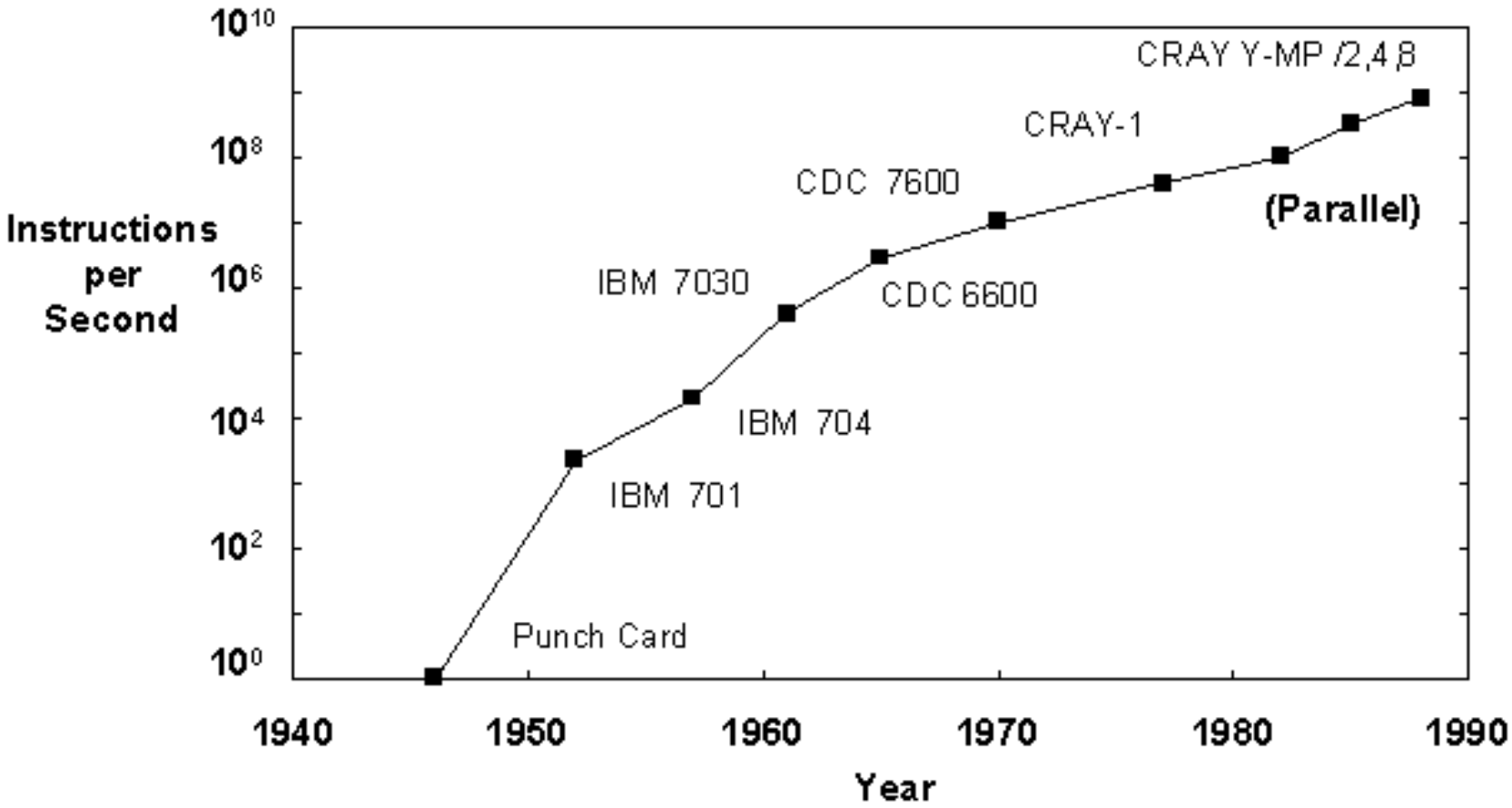
```
For column K = 1 to N-1
  Find the maximum element in column K
    from row K through row N.
  Interchange row K and row P.
  Zero the entry in column K from row K+1 to row N.
```

Sequential Computing and its Limits



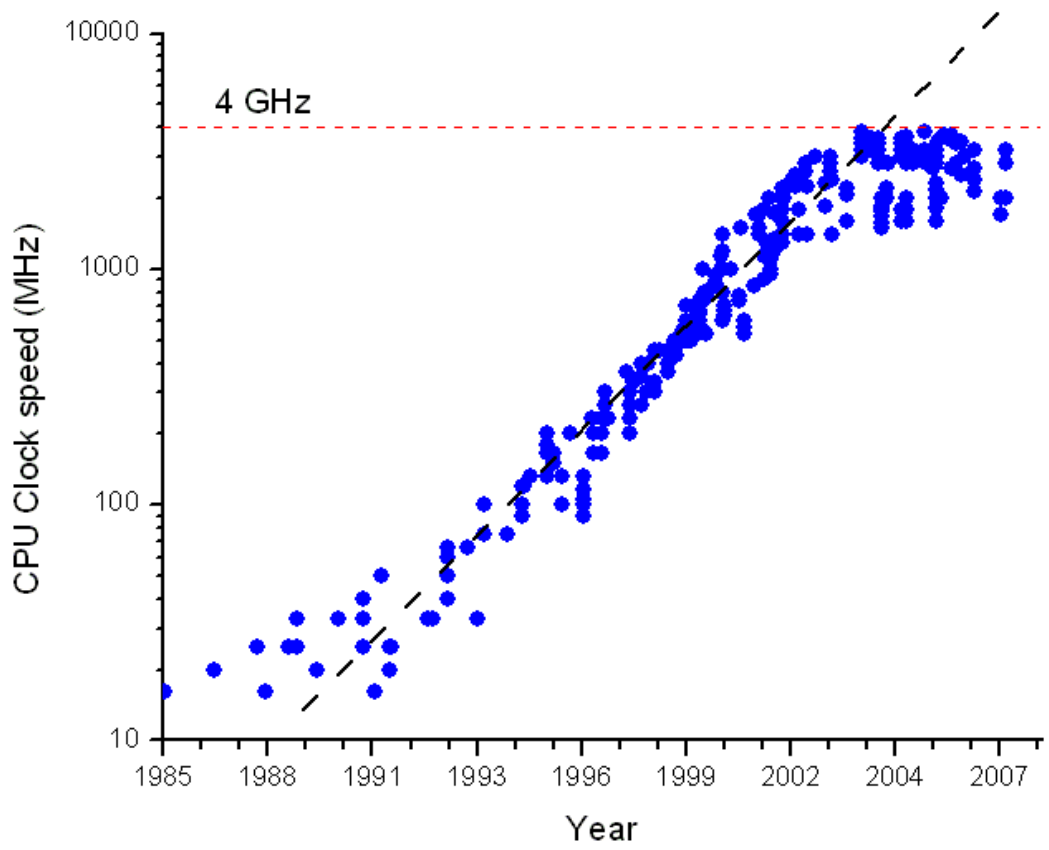
A sequential program computes a sequence of simple tasks in a fixed order, one at a time.

Sequential Computing and its Limits



Improvements in electronics brought faster execution.

Sequential Computing and its Limits



Clock speeds hit a ceiling at 4 GigaHertz because of physical laws (size of atoms, the speed of light, excessive heat.)

Sequential Computing and its Limits

Future processors will have the same clock speed as today...

What do we do for improved performance?

Sequential Computing and its Limits

Hardware possibilities include:

- several, or many, cores on a processor;
- several, or many, processors in one machine;
- tens of machines with very fast communication;
- hundreds or thousands of machines with moderately fast communication.

Sequential Computing and its Limits

Software possibilities include:

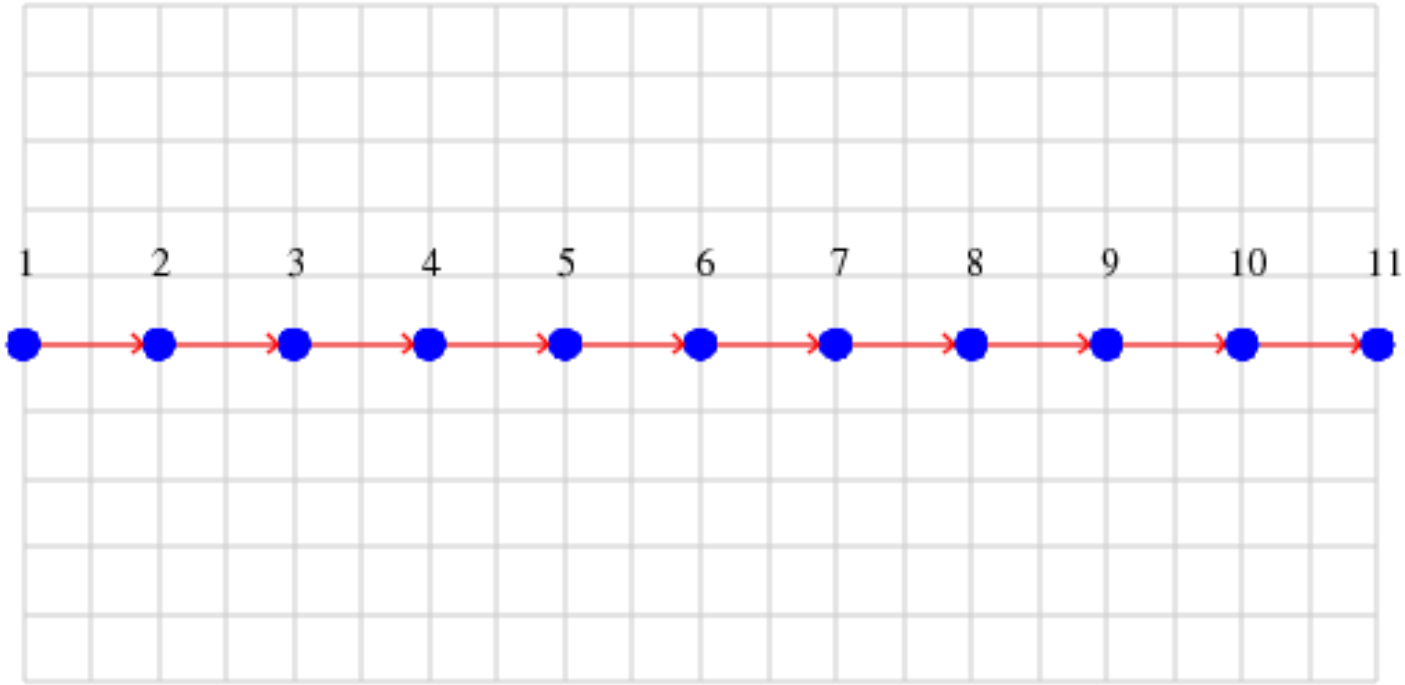
- System software to control multiple cores or processors
- Software to allow separate machines to communicate
- Compilers that extend common languages to control multiple processes
- Algorithms chosen to exploit any inherent parallelism in a problem

Parallel Programming Concepts

1. Sequential Computing and its Limits
2. **Data Dependence**

How many tasks can we do at the same time?

Data Dependence



Tasks are ordered sequentially because a sequential computer can only do one at a time. But is it **logically necessary**?

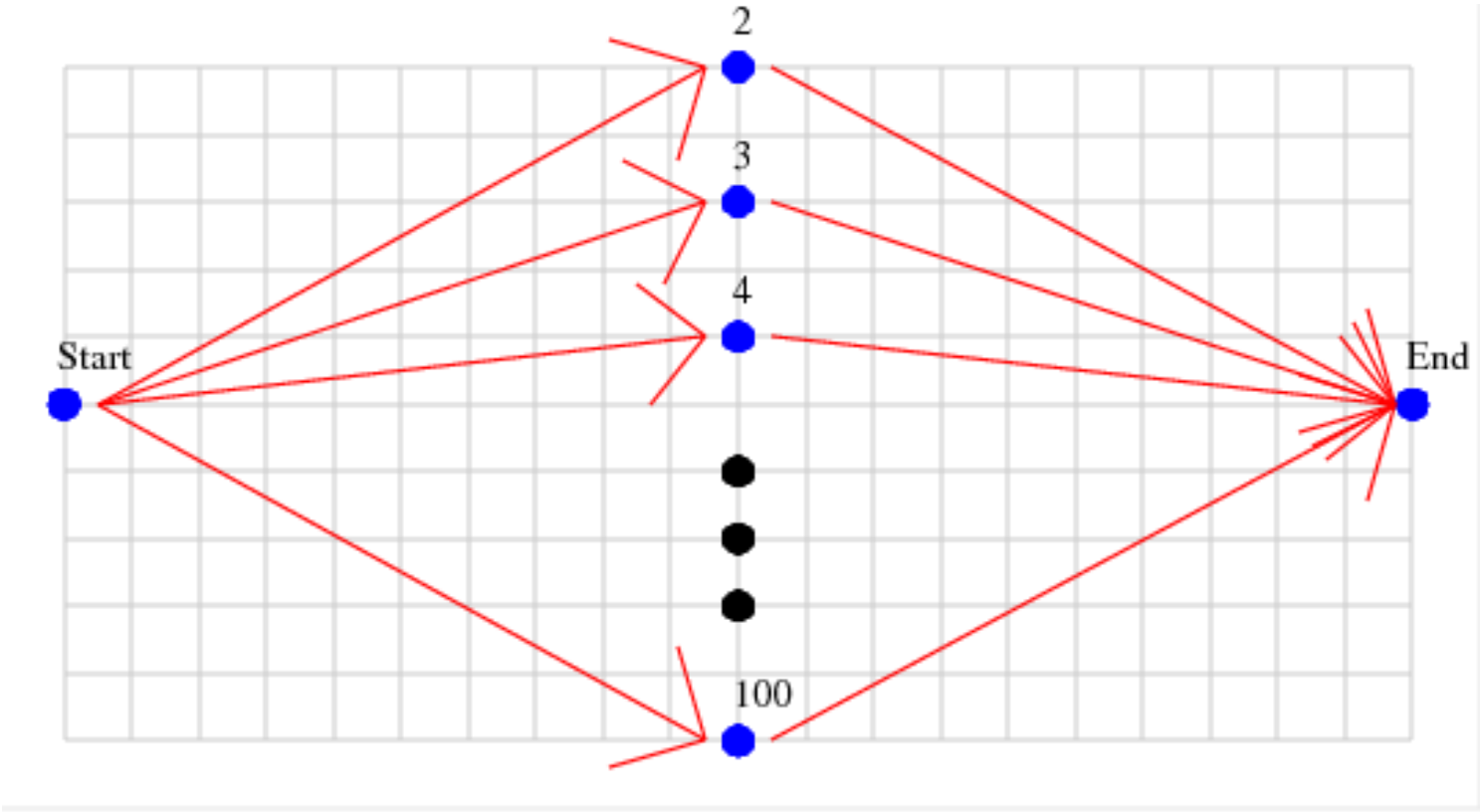
Data Dependence

Computational biologist Peter Beerli has a program named **MIGRATE** which infers population genetic parameters from genetic data using maximum likelihood by generating and analyzing random genealogies.

His computation involves:

1. an input task
2. *thousands* of genealogy generation tasks.
3. an averaging and output task

Data Dependence



In an **embarrassingly parallel** calculation, there's a tiny amount of startup and wrapup, and in between, complete independence.

Data Dependence

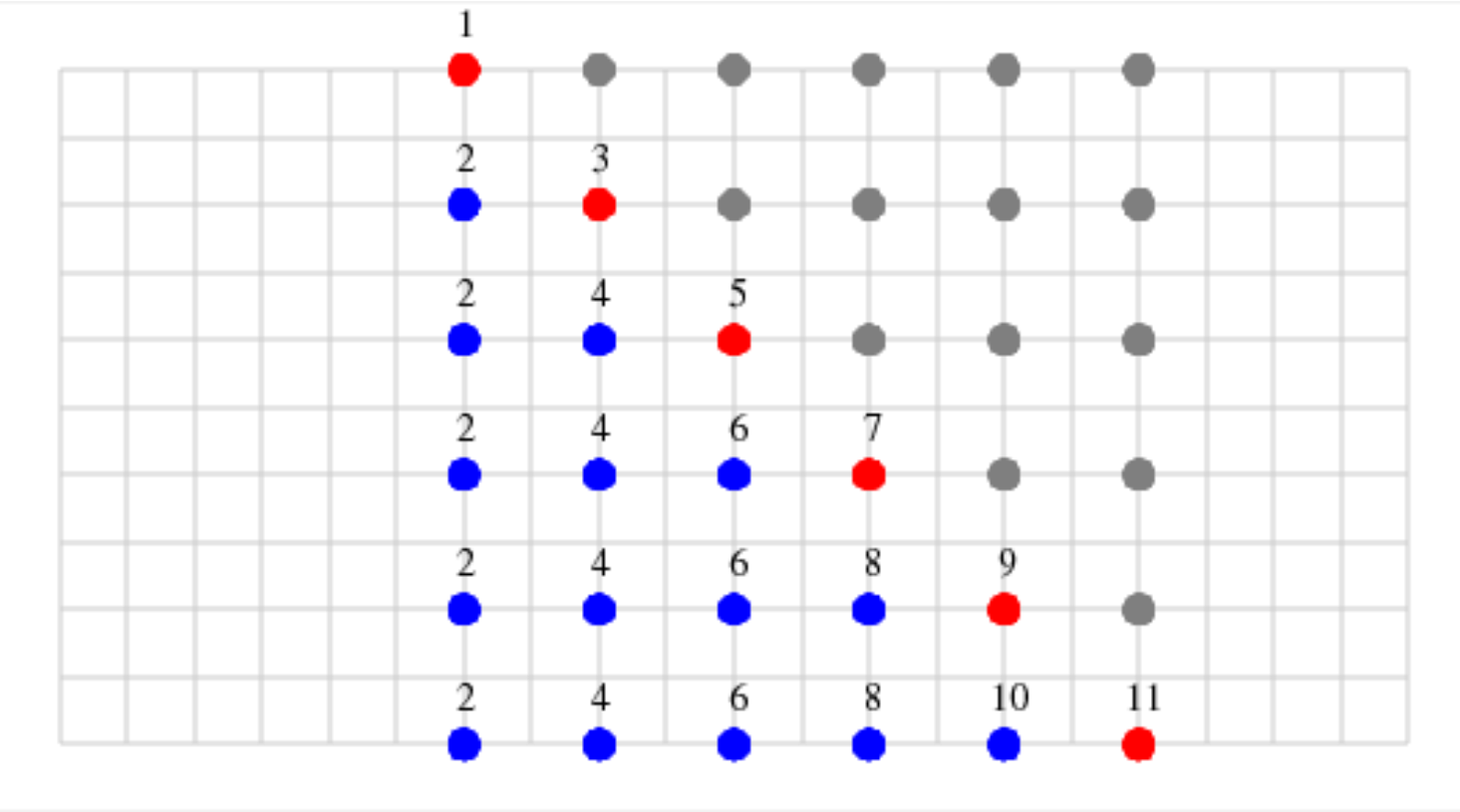
A more typical situation occurs in Gauss elimination of a matrix. Essentially, the number of tasks we have to carry out is equal to the number of entries of the matrix on the diagonal and below the diagonal.

A *diagonal task* seeks the largest element on or below the diagonal.

A *subdiagonal task* adds a multiple of the diagonal row that zeroes out the subdiagonal entry.

Tasks are ordered by column. For a given column, the diagonal task comes first. Then all the subdiagonal tasks are independent.

Data Dependence



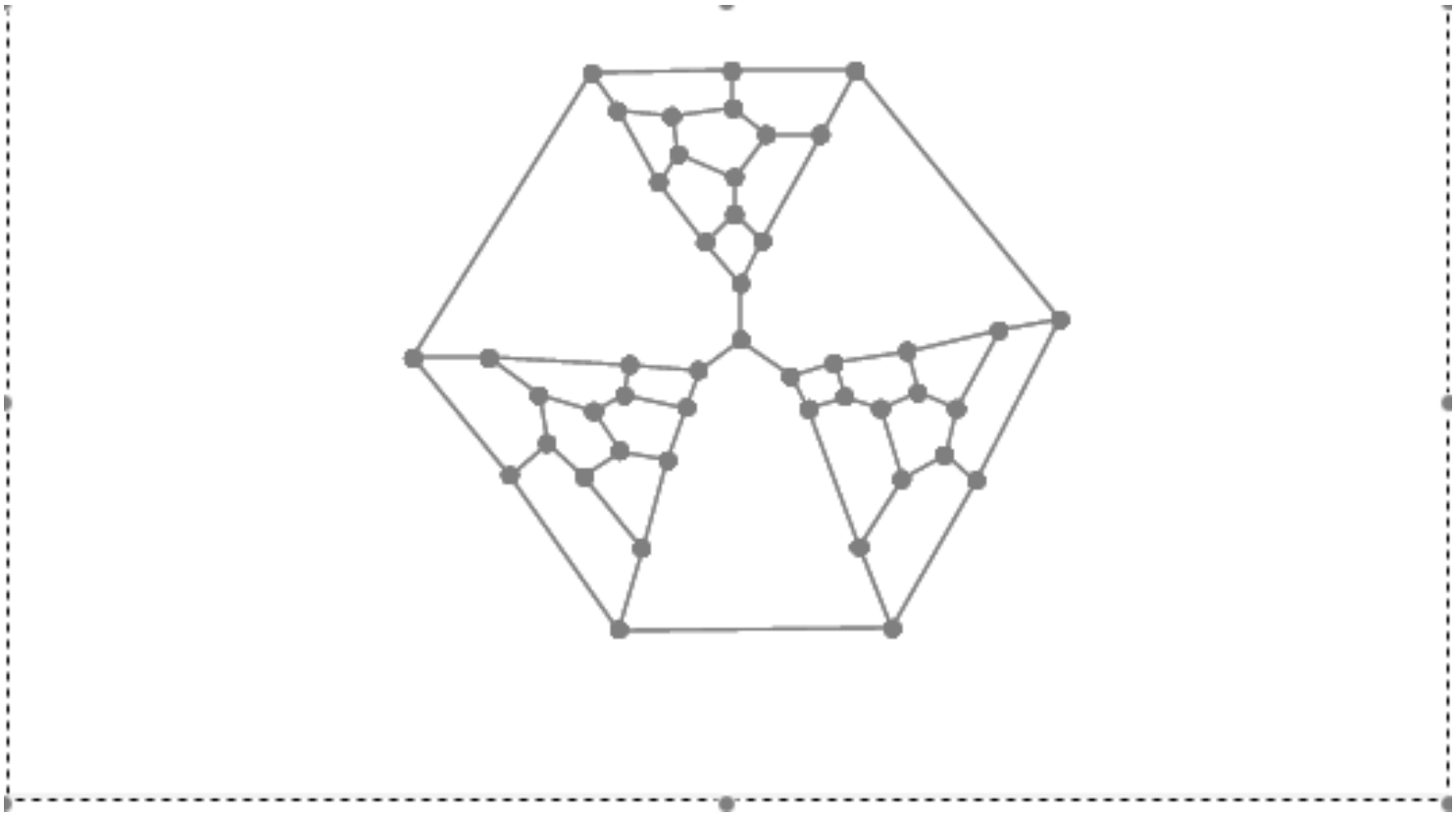
In Gauss elimination, the number of independent tasks available varies from step to step.

Data Dependence

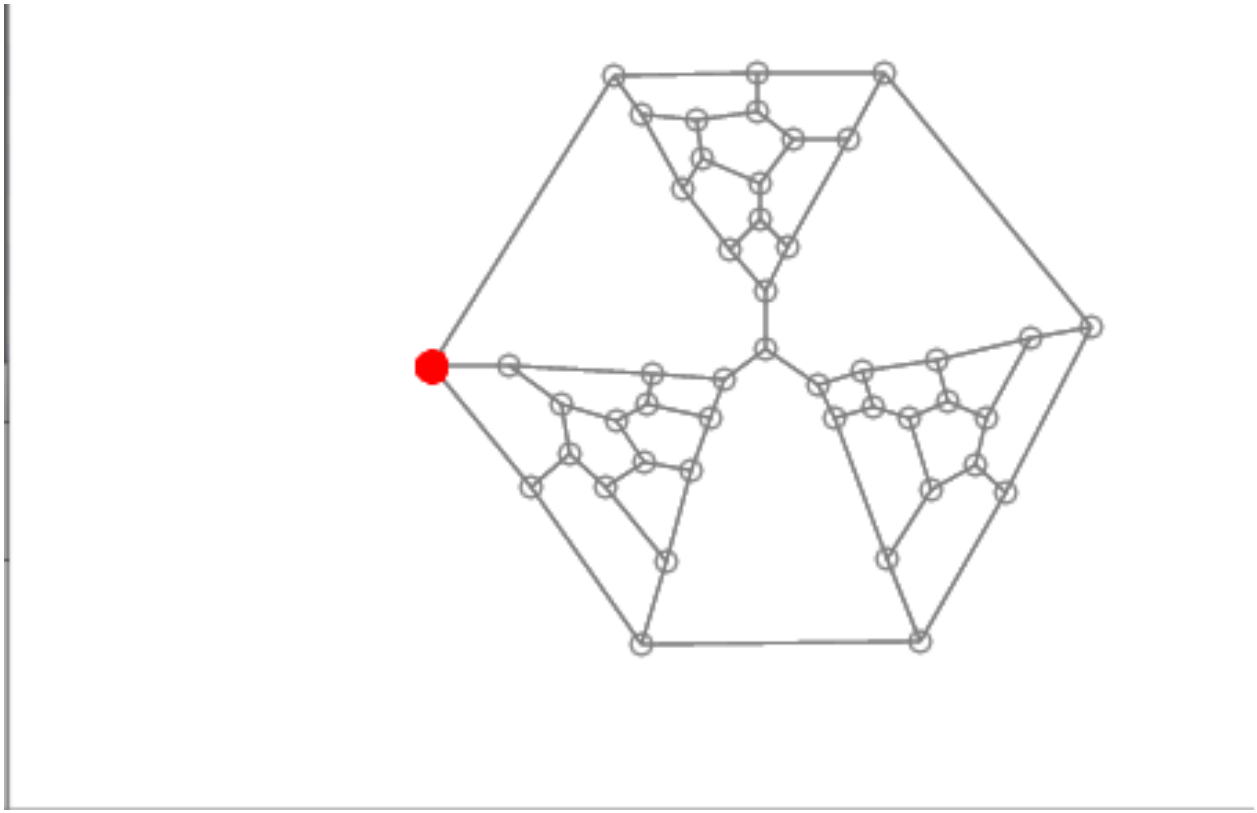
An example that is less structured occurs whenever the computation can be regarded as "exploring" a graph, that is, starting at one root node and following the edges to visit all the nodes.

If the tasks depend on each other in the same way that the nodes are connected back to the root node, then the dependence graph is the tree that starts at the root node.

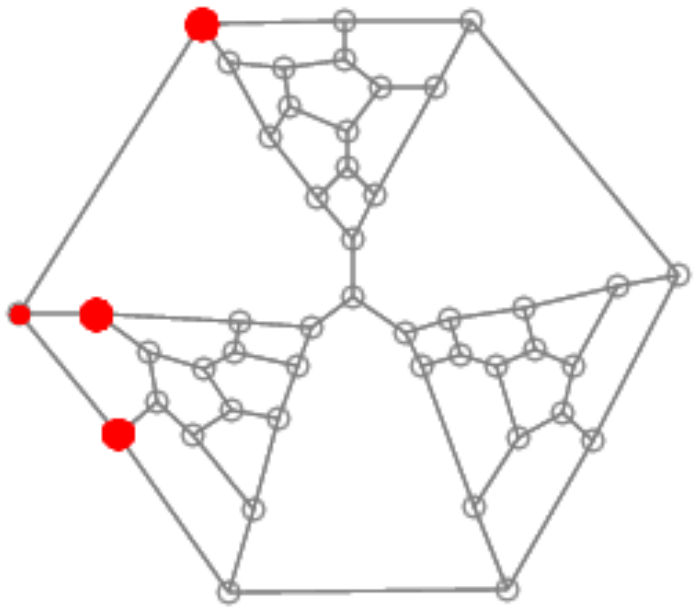
Data Dependence



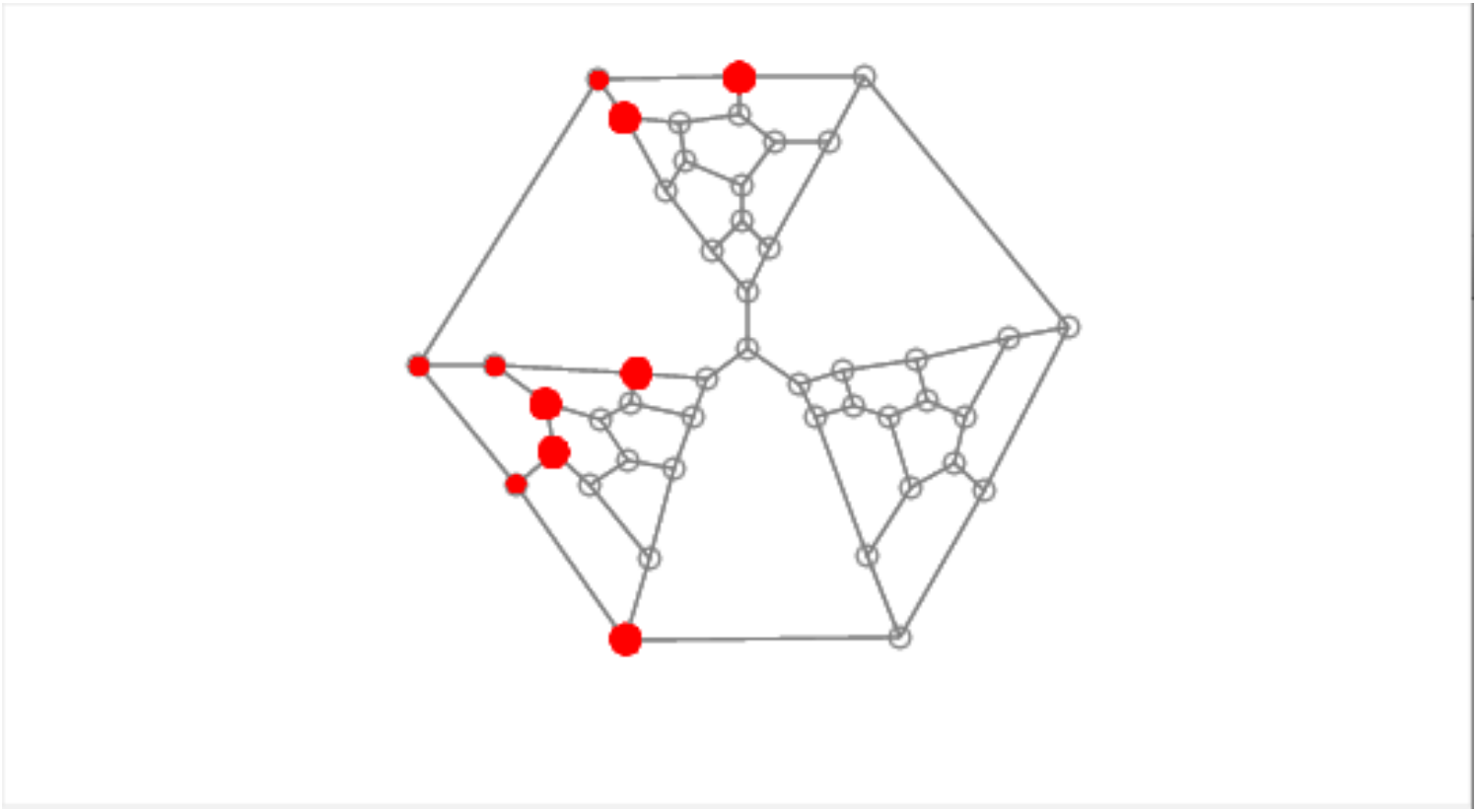
Data Dependence



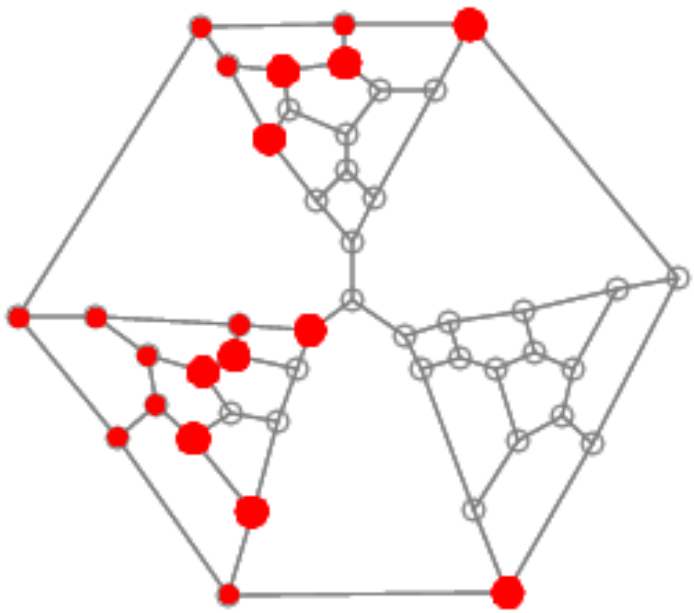
Data Dependence



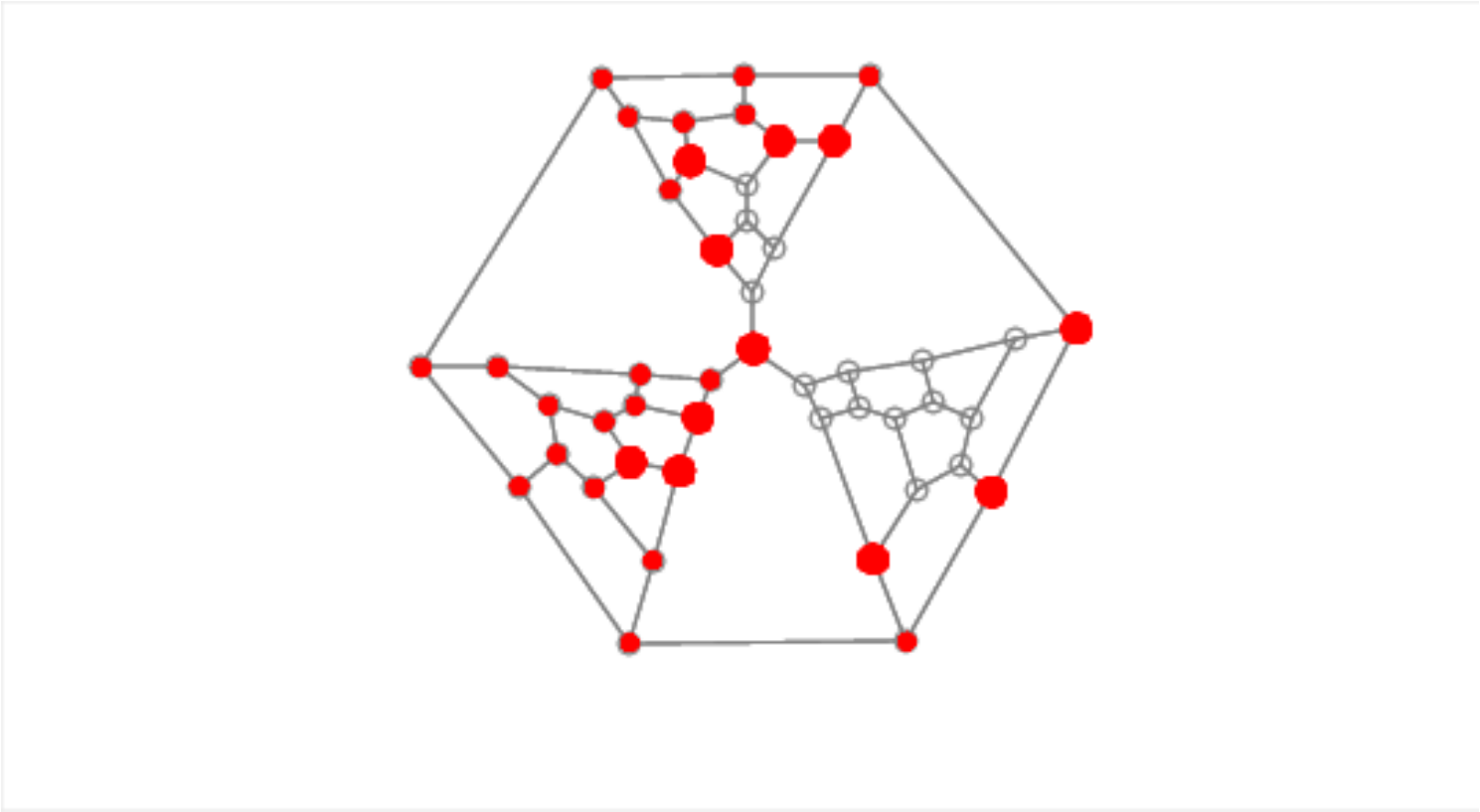
Data Dependence



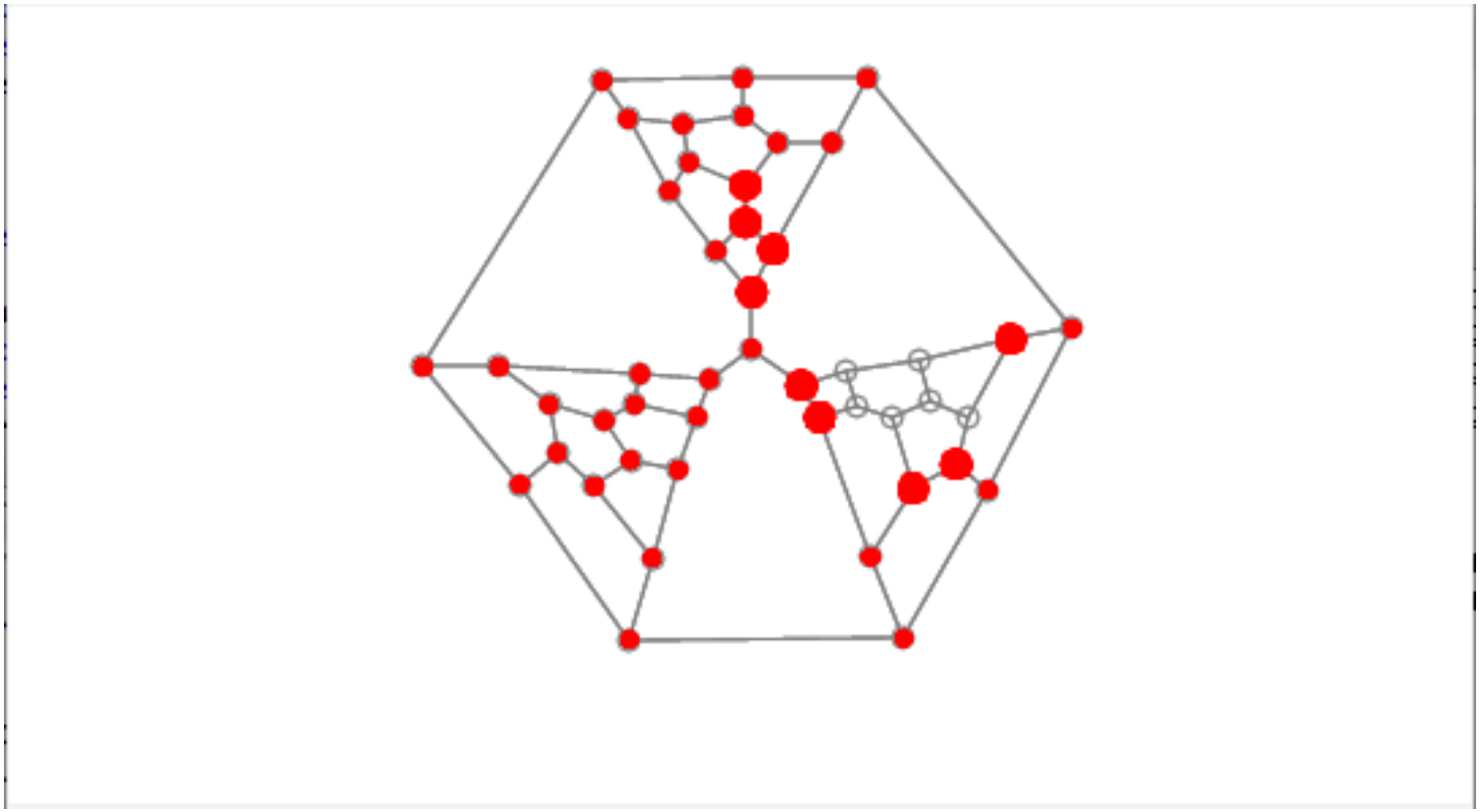
Data Dependence



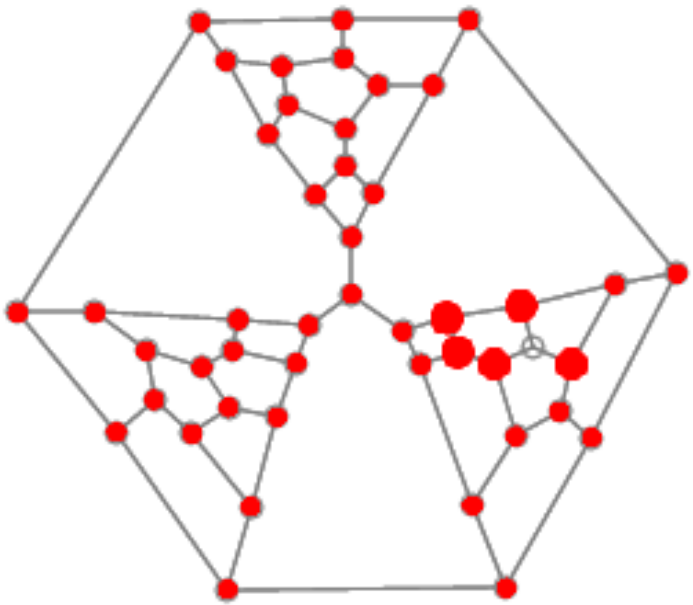
Data Dependence



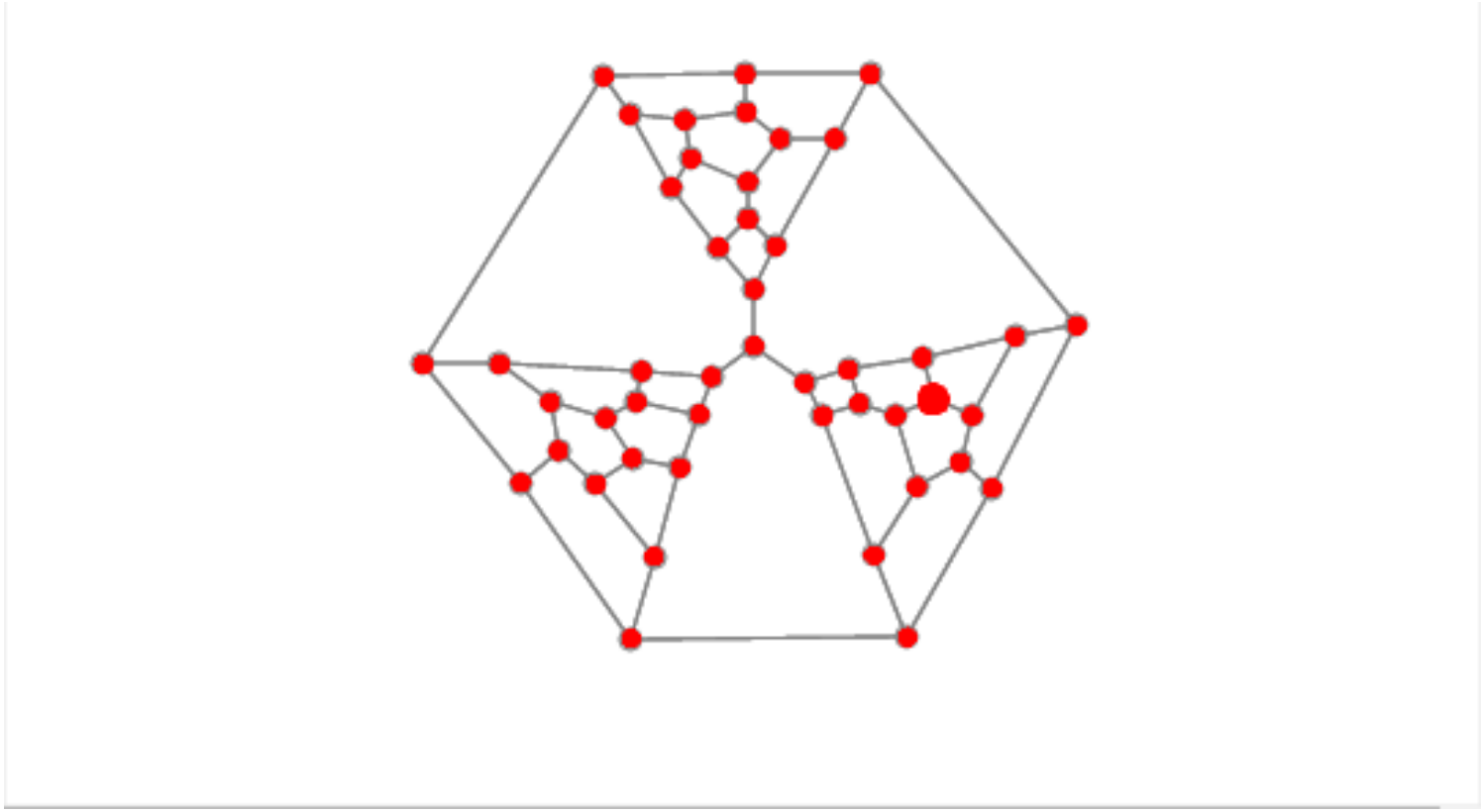
Data Dependence



Data Dependence



Data Dependence



Data Dependence

In our example, we saw the graph, so we know what to expect. In a real problem, the graph might be much more irregular, and we would never see it.

If we are working in parallel, and can do **P** tasks on a step, then at each step, we are completing anywhere from 1 to **P** tasks.

And by completing these tasks, we encounter an unknown number of new tasks that are ready to be worked on.

A sequential program uses a stack to work through such tasks.

Data Dependence - Vocabulary

If the tasks we are considering are relatively large computations, we speak of **coarse grained parallelism**. Such tasks can often be distributed across multiple computers, since the amount of communication (when tasks begin and end) will be relatively small compared to the amount of computation done.

If these tasks are on the order of a statement or a block of statements, we speak of **fine grained parallelism**. Here we presume the tasks will all be done on one computer, but perhaps by different co-processors which share the memory.

Data Dependence

Data dependence between tasks **T1** and **T2**, involving variable **V**, occurs if either:

1. **V** is an input to one task and an output of the other.
2. **V** is an output of both tasks

Data dependence means **T1** and **T2** cannot be performed in parallel.

Data Dependence

If tasks **T1** and **T2** are sets of computer statements, then dependence occurs if one task's "left hand side" variable **V** occurs *anywhere* in the other task (on the left or right hand side).

Arrays, pointers and functions will complicate this test!

Data Dependence

What pairs of tasks can be run in parallel?

```
  a = f + 5              Task 1
  b = sqrt ( w )
--------------------
  e = 2 * e + 1          Task 2
  f = b + d * e
--------------------
  c = f * f - d          Task 3
  d = min ( d, c )
--------------------
  a = 1                  Task 4
  c = 2
```

Data Dependence

For which loops are the iterations data dependent?

```
  for ( i = 1; i < n - 1; i++ )        Loop 1
    x[i] = x[i] + x[i-1];
--------------------
  do j = 2, n - 1
    y(j) = ( y(j) - y(j+1) ) / 2       Loop 2
  end do
--------------------
  for k = 2 : n - 1                    Loop 3
    z(k) = z(k) * z(k);
  end
```

Parallel Programming Concepts

1. Sequential Computing and its Limits
2. Data Dependence
3. **Parallel Algorithms**

We try to understand how parallelism could help us.

Parallel Algorithms

There is a lot we need to understand, but we have to start somewhere!

Let's consider how certain problems could be treated if some kind of parallel computing facility was available.

We won't worry about the details, but we will try to pay attention to some common themes.

Parallel Algorithms

At the end of the year, a teacher has 1,000 test scores to add up and average. Her noisy classroom of 50 kids is distracting her.

She comes up with several ways to use their help.

Method 1

The students could all come up to the front desk and stare at the gradebook with the 1,000 scores.

Each student could take one row of the table of scores and add it up.

They only have one sheet of paper to work on, so the teacher would mark it off into 50 separate boxes for their intermediate sums.

The teacher would pencil in a 0 for the initial sum. Students take turns erasing the current sum, adding their result, and writing the new sum.

Method 2

The teacher could hand each student one set of grades to add.

Each student work at their desk, without worrying about any interference from others.

As soon as they have computed their sum, they call the teacher over, who adds the result to the total.

Parallel Algorithms

These two simple stories might seem very similar, but they suggest two different ways to go about a parallel algorithm; we will come back to these ideas soon.

But let us take from this exercise a simple model for a parallel computation.

Let's say there's a **Boss** or **Master**, with **N** pieces of data, typically **X1, X2, ...XN**.

We let **p** stand for the number of **helpers** or **agents**, each of whom works with **n** pieces of data, typically **x1, x2,... xn**.

Parallel Algorithms

To sum **N** numbers, the Master sends **n** numbers to each agent.

When an agent returns the partial sum, the Master performs the final reduction of the partial sums to a single total.

The same amount of computation is done, and happens faster because the big sum has been broken up.

But this computation may take much longer. For every computation (plus sign) there is also a communication (send the number). Communication between machines takes much longer than computation (100 or 1,000 times longer).

We will see other examples of how communication costs are an important item in judging a parallel algorithm.

Parallel Algorithms

Suppose on the other hand that communication costs are insignificant, and that computing power is so cheap that we have one agent for every number **Xi** in our sum.

Suppose that an agent can add two numbers in one time step, and that a sequential program would take about **N** steps to compute the sum.

Then we can add **N** numbers in **log(N)** time using our agents.

We start with each agent having one number.

On the first step, each even agent **I** sends its value to the agent **I/2**, who adds it to its number.

The even agents drop out, we renumber the odd agents, and repeat the process til we end up with the total sum stored by agent 1.

Parallel Algorithms

The addition examples may seem unrealistic and impractical.

But the pattern of combination used by these simple tasks often occurs in more complicated calculations. In those cases, a similar pattern of data distribution and combination might turn out to be appropriate and efficient.

Parallel Algorithms

Now suppose we have a set of **N** numbers to sort.

You probably know several algorithms for sorting, but they are all expressed sequentially.

Suppose we use **N/2** agents, giving each two values, **y1** and **y2**.

We number the agents, and imagine them standing in a line, able to communicate with their left and right neighbors.

The following algorithm sorts the data in **N** steps. The best sequential algorithms take about **N log(N)** steps.

Parallel Algorithms

During a step, each agent:

```
sorts its two values so that y1 < y2,

sends a copy of y1 left;
receives a number x2 from the left.
y1 = max ( y1, x2 );

sends a copy of x2 right.
receives a number z1 from the right.
y2 = min ( y2, z1).
```

Parallel Algorithms

Some features of this algorithm that occur elsewhere:

- the agents are arranged in a communication network
- the communication is regular (a number is passed left, then a number is passed right)
- the communication is local (left or right)

If the number of agents is limited, it's not difficult to modify the algorithm so that each agent handles more than just 2 values.

By the way, what should we do at the first and last agents?

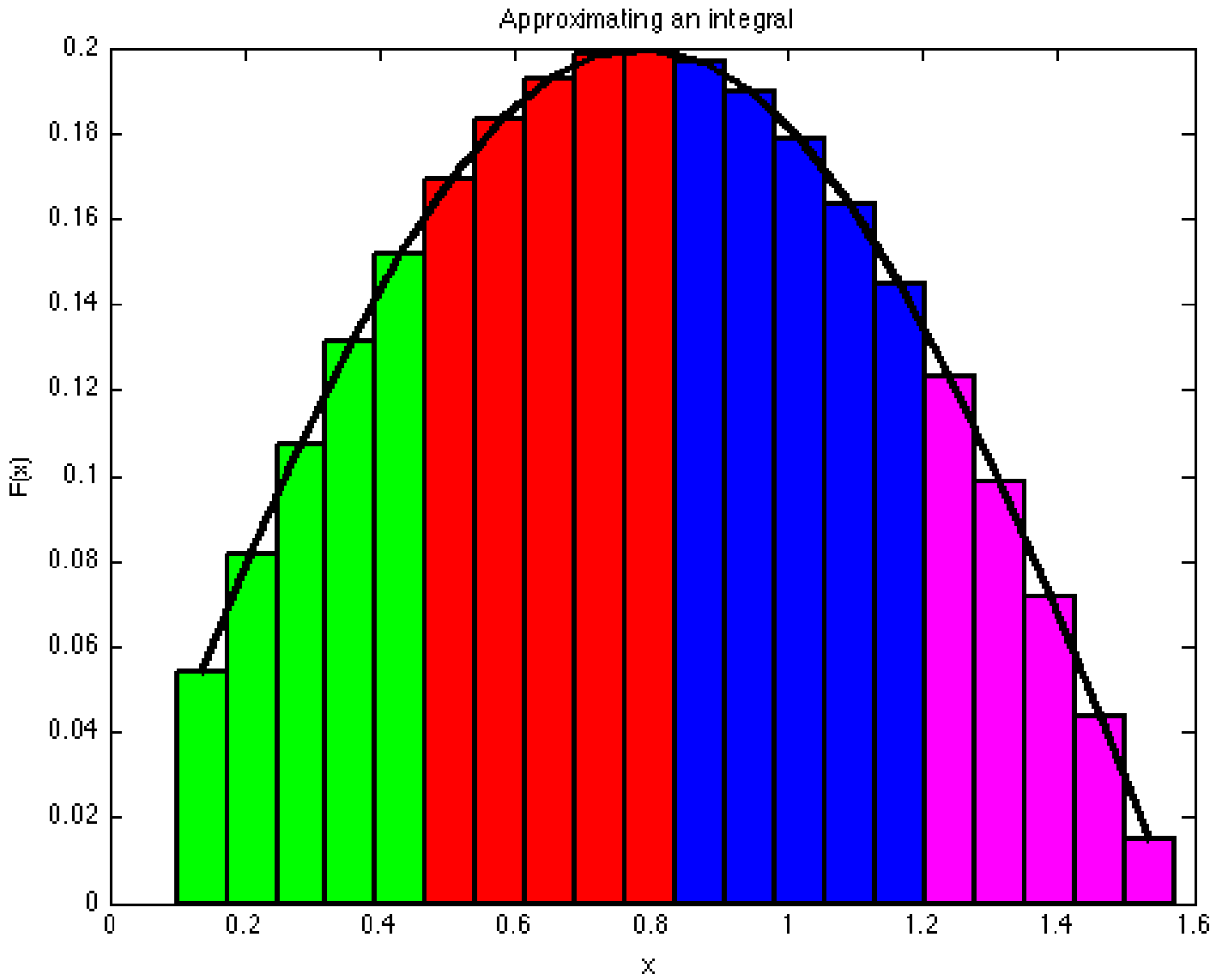
Parallel Algorithms

Quadrature involves the estimation of the integral of a function over a region, usually by averaging a number of sample function values.

In the simplest case, the region is a line, the function is "well behaved", and we know the number of sample values we want.

In that case we can divide the line in segments, and have each agent sample its segment, sum its values and send them to the "Boss" for a final total.

Parallel Algorithms



Parallel Algorithms

When a parallel version of an algorithm is produced by subdividing the geometric region, this technique is known as **domain decomposition**.

In this case, the division was simple. But sometimes, the subregions will need to communicate with each other.

If the function changes behavior, then it may be necessary to do more sampling in some regions. Instead of dividing the whole region into 4 segments, we could have given each agent a part, but left most of the work unassigned. As an agent finishes a task, it gets more work. This is known as **dynamic scheduling**.

Parallel Algorithms

The **power method** seeks the maximum eigenvalue λ of a matrix **A**, and its corresponding eigenvector **v**.

Starting with an initial estimate for **v**, we essentially do the following:

```
v_new = A * v;
lambda = v_new' * v;
v = v_new / || v_new ||;
```

Matrix-vector multiplication is easy to do in parallel. We can divide **A** into **p** groups of rows, or columns, or even into submatrices. Since the matrix doesn't change, the communication at each step involves sending pieces of **v** and **v_new** back and forth.

Parallel Algorithms

Parallel Algorithms

I mentioned the idea of searching a graph or tree. The solution of a Sudoku puzzle is an illustration of this concept. As you can see, the partially-filled in puzzle includes indicators for the digits that are possible fillers for the empty boxes.

Every digit represents a task, that is, moving to a new node on the tree and taking yet another (educated) guess.

Clearly, a set of agents could profitably explore different possibilities.

Note that in this example, two agents could start out in different directions and end up exploring the same node further down the tree.

This is a problem whose "geometry" is very hard to diagram in advance.

Parallel Algorithms

The heat equation is a model of the kinds of partial differential equations that must be solved in scientific computations.

Determine the values of $H(x, t)$ over a range $t_0 <= t <= t_1$ and space $x_0 <= x <= x_1$, given an initial value $H(x, t_0)$, boundary conditions, a heat source function $f(x, t)$, and a partial differential equation

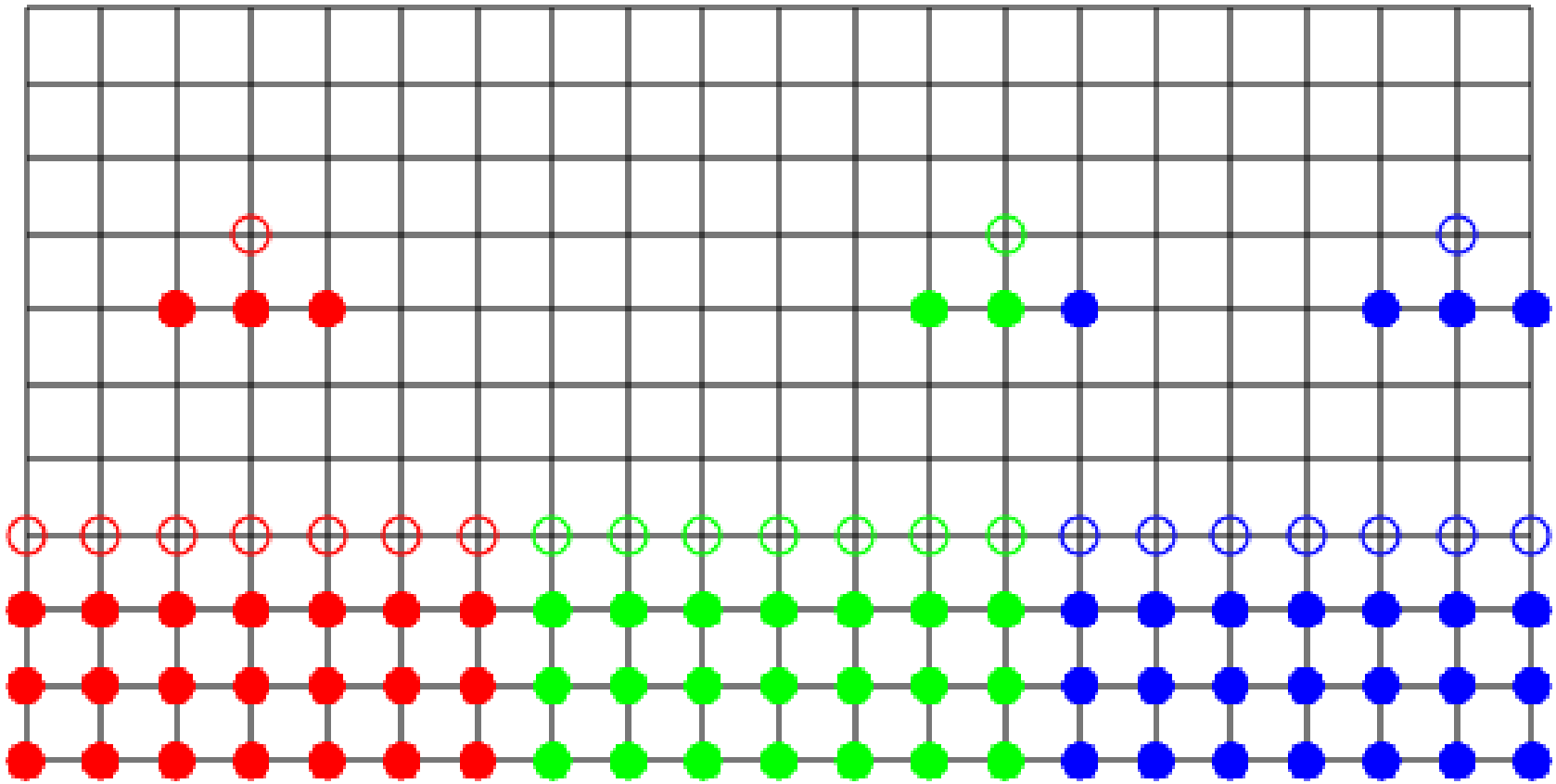
$$\frac{\partial H}{\partial t} - k\frac{\partial^2 H}{\partial x^2} = f(x, t)$$

Parallel Algorithms

An approximate solution to this problem can be computed by using a grid of points in space and time. The values are known at the initial time, and at the left and right endpoints. Internally, we can compute a value at the next time by a kind of average of the current value with its left and right neighbors.

We can use domain decomposition, as we did for the quadrature problem. However, we now see that we will need to arrange for communication between neighboring agents, since in some cases, the neighbor of a point belongs to a different subregion.

Parallel Algorithms



Parallel Programming Concepts

1. Sequential Computing and its Limits
2. Data Dependence
3. Parallel Algorithms
4. **Shared Memory Parallel Computing**

A shared memory program runs on one computer with multiple cores.

Shared Memory - Multiple Processors

The latest CPU's are called *dual core* and *quad core*, with rapid increases to 8 and 64 cores to be expected.

The cores share the memory on the chip.

A single program can use all the cores for a computation.

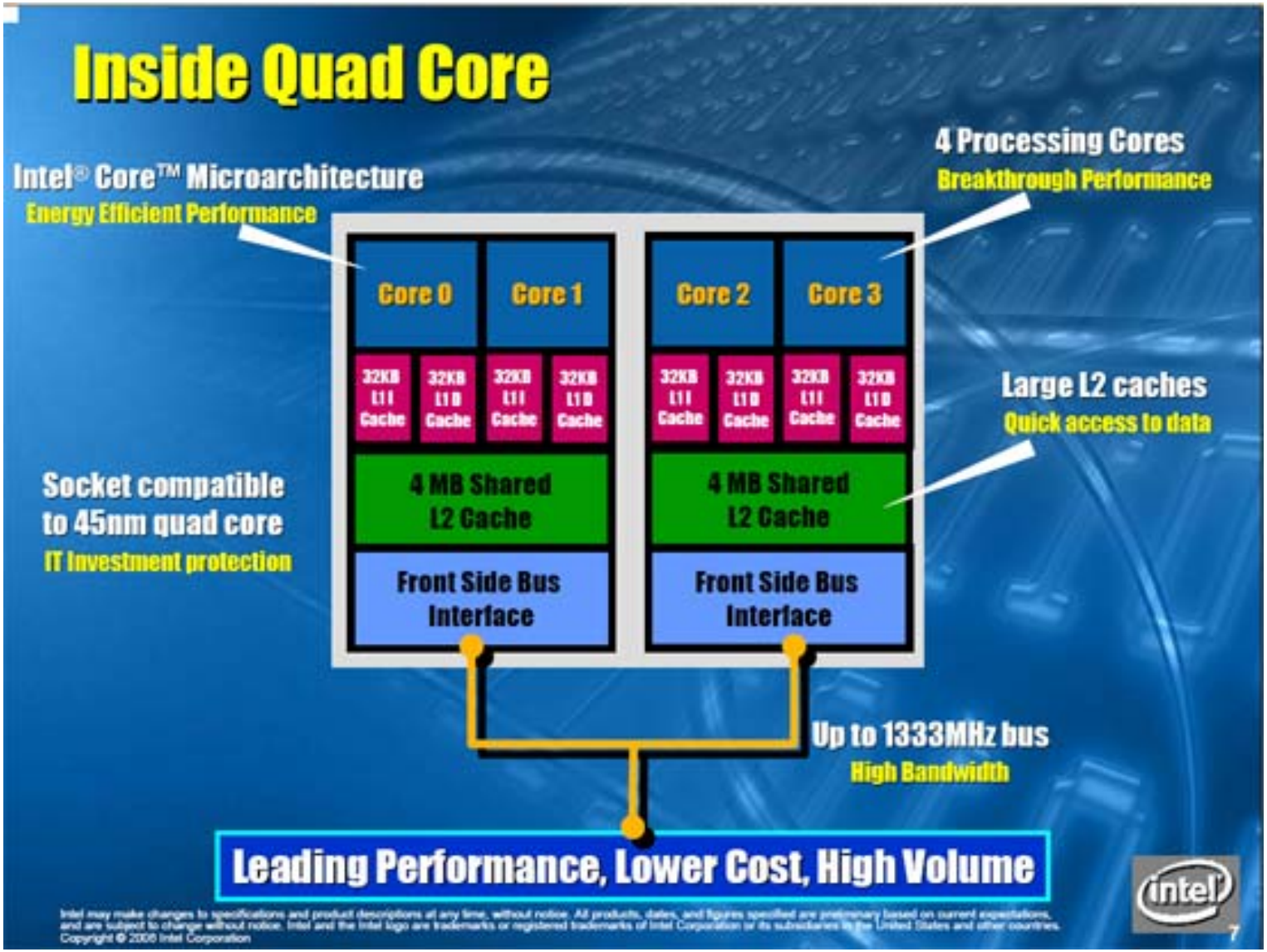
It may be confusing, but we'll refer to a core as a *processor*.

Shared Memory - Multiple Processors

If your laptop has a dual core or quad core processor, then you will see a speedup on many things, simply because the operating system can run different tasks on each core.

When you write a program to run on your laptop, though, it probably will *not automatically* benefit from multiple cores.

Shared Memory - Multiple Processors



Shared Memory - Multiple Local Memories

The diagram of the Intel quad core chip includes several layers of memory associated with each core.

The full memory of the chip is relatively "far away" from the cores. The cache contains selected copies of memory data that is expected to be needed by the core.

Anticipating the core's memory needs is vital for good performance. (There are some ways in which a programmer can take advantage of this.)

A headache for the operating system: *cache coherence*, that is, making sure the original data is not changed by another processor, which invalidates the cached copy.

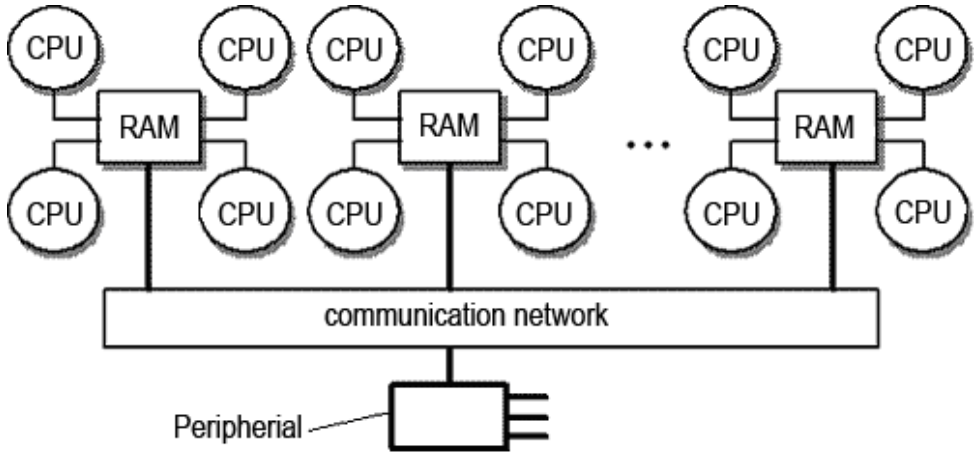
Shared Memory - NUMA Model

It's easy for cores to share memory on a chip. And each core can reach any memory item in the same time, known as **UMA** or "Uniform Access to Memory".

Until we get 8 or 16 core processors, we can still extend the shared memory model, if we are willing to live with **NUMA** or "Non-Uniform Access to Memory".

We arrange several multiprocessor chips on a very high speed connection. It will now take longer for a core on one chip to access memory on another chip, but not too much longer, and the operating system makes everything look like one big memory space.

Shared Memory - NUMA Model



Chips with four cores share local RAM, and have access to RAM on other chips.

VT's SGI ALTIX systems use the NUMA model.

Shared Memory - Implications for Programmers

On a shared memory system, the programmer does not have to worry about distributing the data. It's all in one place *or at least it looks that way!*

A value updated by one core must get back to shared memory before another core needs it.

Some parallel operations may have parts that only one core at a time should do (searching for maximum entry in vector).

Parallelism is limited to the number of cores on a chip (2, 4, 8?), or perhaps the number of cores on a chip *multiplied* by the number chips in a very fast local network (4, 8, 16, 64, ...?).

Shared Memory - Implications for Programmers

The standard way of using a shared memory system in parallel involves **OpenMP**.

OpenMP allows a user to write a program in C/C++ or Fortran, and then to mark individual loops or special code sections that can be executed in parallel.

The user can also indicate that certain variables (especially "temporary" variables) must be treated in a special way to avoid problems during parallel execution.

The compiler splits the work among the available processors.

This workshop will include an introduction to **OpenMP** in a separate talk.

Shared Memory - Data Conflicts

Here is one example of the problems that can occur when working in shared memory.

Data conflicts occur when data access by one process interferes with that of another.

Data conflicts are also called *data races* or *memory contention*. In part, these problems occur because there may be several copies of a single data item.

If we allow the "master" value of this data to change, the old copies are out of date, or *stale data*.

Shared Memory - Data Conflicts

A mild problem occurs when many processes want to **read** the same item at the same time. This might cause a slight delay.

A bigger problem occurs when many processes want to **write** or modify the same item. This can happen when computing the sum of a vector, for instance. But it's not hard to tell the processes to cooperate here.

A serious problem occurs when a process does not realize that a data value has been changed, and therefore makes an incorrect computation.

Data Conflicts: The VECTOR MAX Code

```
program main

integer i, n
double precision x(1000), x_max

n = 1000

do i = 1, n
  x(i) = rand ( )
end do

x_max = -1000.0

do i = 1, n
  if ( x_max < x(i) ) then
    x_max = x(i)
  end if
end do

stop
end
```

Shared Memory - Data Conflicts - VECTOR MAX

```
      program main

      include 'omp_lib.h'

      integer i, n
      double precision x(1000), x_max

      n = 1000

      do i = 1, n
        x(i) = rand ( )
      end do

      x_max = -1000.0

!$omp parallel do
      do i = 1, n
        if ( x_max < x(i) ) then
          x_max = x(i)
        end if
      end do
!$omp end parallel

      stop
      end
```

Shared Memory - Data Conflicts - VECTOR MAX

It's hard to believe, but the parallel version of the code is incorrect. In this version of an OpenMP program, the variable **X_MAX** is *shared*, that is, every process has access to it.

Each process will be checking some entries of **X** independently.

Suppose process P0 is checking entries 1 to 50, and process P1 is checking entries 51 to 100.

Suppose X(10) is 2, and X(60) is 10000, and all other entries are 1.

Shared Memory - Data Conflicts - VECTOR MAX

Since the two processes are working simultaneously, the following is a possible history of the computation:

1. **X_MAX** is currently 1.
2. P1 notes that **X_MAX** (=1) is less than **X**(60) (=10,000).
3. P0 notes that **X_MAX** (=1) is less than **X**(10) (=2).
4. P1 updates **X_MAX** (=1) to 10,000.
5. P0 updates **X_MAX** (=10,000) to 2.

and of course, the final result **X_MAX**=2 will be wrong!

Shared Memory - Data Conflicts - VECTOR MAX

This simple example has a simple correction, but we'll wait until the **OpenMP** lecture to go into that.

The reason for looking at this problem now is to illustrate the job of the programmer in a shared memory system.

The programmer must notice points in the program where the processors could interfere with each other.

The programmer must coordinate calculations to avoid such interference.

Parallel Programming Concepts

1. Sequential Computing and its Limits
2. Data Dependence
3. Parallel Algorithms
4. Shared Memory Parallel Computing
5. **Distributed Memory Parallel Computing**

Distributed memory programs run on multiple computers; messages are used to send data and results.

Distributed Memory - Multiple Processors

Distributed Memory

A distributed memory computation manages the resources of several independent computers which are joined on a communication network.

Thus, theoretically, no fancy new hardware is needed: just cables and a router.

If the communication is fast enough, the result is a powerful computing device.

Distributed Memory

A distributed memory system will need some kind of software daemon, running on each machine, which will:

- copy the program and data files to all machines;
- start the program on all the machines;
- send and receive messages from programs, holding them til received;
- gather program outputs to a single file
- shut down programs at the end and clean up

Distributed Memory

A distributed memory system must offer a library that allows a user program to make function calls:

- to "check in"
- to find out how many other processes are running
- to request an ID
- to send a message to another process
- to receive messages from other processes
- to request that other processes wait
- to "check out"

Distributed Memory

The programmer writes a single program to run on all machines.

This program

- defines and initializes data
- determines the portion of work that a process will do
- shares data with other processes as needed
- collects results in the end.

Distributed Memory

How can copies of the same program execute differently on different machines?

Each process is assigned a unique **ID**, which it can find out through a function call.

Each process can also find out **P**, the total number of processes participating in the calculation.

ID's run from 0 to **P-1**. Simply assigning an ID is enough to guide the processes to the work they must do.

Distributed Memory

A simple design for parallel programs is the **master/worker** model.

Process 0 is assigned to control the computation: read input, assign work, gather results, print reports.

Independent tasks are divided up among processes 1 through **P-1**.

People like this model because it is not so different from sequential programming - someone is still "in charge".

IF/ELSE statements separate the master and worker portions of the program.

Distributed Memory

Sketch of a master/worker quadrature program:

```
if ( master )
  send N to all workers
  for each worker I, send A[I], B[I]
  set Q = 0
  for each worker I, receive Q_PART[I], add to Q.
  print "Integral = " Q
else
  receive N from master
  receive A, B from master
  Q_PART = 0
  for N equally spaced X between A and B,
    Q_PART = Q_PART + F(X)
  send Q_PART to master
```

Distributed Memory

Another design for parallel programs comes from domain decomposition.

If the problem to be solved is associated with a geometric region, then process **I** is assigned to work in subregion **I**.

Communication between processes corresponds to the geometry of the subregions. For the 1D heat equation, each process (except the first and last) has a left and right neighbor.

Typically, a process must inform its neighbors of data it has updated that lies on their common boundary.

The amount of communication is typically related to the "area" of the boundary. It's useful to keep the subregions compact.

Distributed Memory

Here is a sketch of the heat equation program:

```
Process ID is assigned interval [ID/P, (ID+1)/P].
Nodes: X[0], X[1] through X[N], X[N+1].
Initialize: U[0], U[1] through U[N], U[N+1].

Time Loop:
  Compute new values of U[1] through U[N].
  Send U[1] to left, get U[0] from left.
  Send U[N] to right, get U[N+1] from right.

Print solution at end:

  U[1] through U[N] from ID 0,
  U[1] through U[N] from ID 1, ...
```

Distributed Memory

Using distributed memory brings some new costs and potential errors.

The costs may include a substantial effort to rethink and rewrite the program.

There is also a communication cost that offsets the; remember we suggested that sending a number somewhere else might take 100 or 1000 times as long as performing a computation on that number.

But there are also several new problems that can occur because of the fact that we are trying to communicate between machines.

Distributed Memory

Generally, the processes are free to run independently on the various machines, as fast as they can.

In most programs there will be certain **synchronization points**, that is, lines of code which some or all of the processors must reach at the same time.

Processors reaching the synchronization point early must wait for the others.

As an example, if the processors are working together on an iteration, then generally they all must synchronize at the end of each step, to share updated information.

Distributed Memory

Synchronization can increase the costs and delays associated with communication.

In particular, if the work is poorly divided, or if communication to a particular processor is slow, or if that processor is less powerful, then it is likely that all the other processors will have to wait at synchronization points.

Problems of this sort are called **load balancing problems**.

Distributed Memory

Another instance that can involve synchronization involves the sending of messages.

In the typical case, only two processes are involved, a sender and a receiver.

In the simplest case, the sender pauses at the **SEND** statement, the receiver pauses at the **RECEIVE** statement. When they are both there, the sender sends the message, the receiver receives it, and sends back a confirmation.

The idle time that one process spent waiting for the other (a synchronization point) is wasted.

Distributed Memory

There are alternatives when transmitting messages:

1. *Synchronous transmission*: the message is not transmitted until both sender and receiver are ready;
2. *Buffered transmission*: the message is put into a buffer as soon as the sender is ready. The sender proceeds, and the receiver picks up the message when it is ready
3. *Nonblocking transmission*: the receiver indicates that it is ready to receive the message, but may continue to do other work while waiting

Distributed Memory

Not only can message transmission cause delays - it can also cause the program to fail entirely.

It is possible to write a program which uses the communication channels incorrectly, so that messages can't get through.

Deadlock is a situation in which a process cannot proceed, because it is waiting for a condition that will never come true.

Recall that in the simplest version of message transmission, a sender can't proceed until the message is received; a receiver can't proceed until the message is sent.

Distributed Memory - Deadlock

Here is a recipe for deadlock, using our heat equation problem.

Suppose we use 2 processes, each responsible for computing 50 values of X. To keep things simple, we will refer to all 100 values of X as though they were in a single global array.

Process P0 has X(1) through X(50)
process P1 has X(51) through X(100).

P0 needs updated copies of X(51) from P1;
P1 needs updated copies of X(50) from P0.

If they do this in the wrong way, they deadlock.

Distributed Memory - Deadlock

```
===The P0 side==============+====The P1 side============
                           |
Initialize X(1)...X(50)    |  Initialize X(51)...X(100).
                           |
Begin loop                 |  Begin loop
                           |
  Send X(50) to P1      ==>|<== Send X(51) to P0
                           |
  When receipt confirmed,<==|==> When receipt confirmed,
                           |
  Receive X(51) from P1 <==|==> Receive X(50) from P0
                           |
  Update X(1)...X(50).     |    Update X(51)...X(100).
                           |
Repeat                     |  Repeat
```

Distributed Memory - Programmer Implications

MPI is a system designed for parallel programming on distributed memory systems.

MPI allows the user to write a program, in C/C++ or Fortran.

The user program calls various functions from the MPI library in order to determine the process ID, send messages, and other tasks.

Depending on the installation, the user runs a job interactively, using a command like **mpirun** or through a batch system, which requires a job script.

The next two days of this workshop will concentrate on a detailed presentation of **MPI**.

Let's get a preview of an **MPI** program.

Distributed Memory: The VECTOR MAX Code

```
program main

include 'mpif.h'

integer id
integer ierr
integer p

call MPI_Init ( ierr )
call MPI_Comm_rank ( MPI_COMM_WORLD, id, ierr )
call MPI_Comm_size ( MPI_COMM_WORLD, p, ierr )

call do_stuff ( id, p )

call MPI_Finalize ( ierr )

stop
end
```

Distributed Memory: The VECTOR MAX Code

```
      subroutine dostuff ( id, p )

      include 'mpif.h'

      integer i, id, ierr, j, p, src, tag, target
      double precision x(1000), x_max, x_max_local

      if ( id == 0 ) then              <--MASTER creates data and sends it

        do i = 1, p - 1

          do j = 1, 1000
            x(j) = rand ( )
          end do

          target = i
          tag = 1
          call MPI_Send ( x, 1000, MPI_DOUBLE_PRECISION, target, tag,
     &      MPI_COMM_WORLD, ierr )

        end do

      else                             <-- WORKERS receive data

        src = 0
        tag = 1
        call MPI_Recv ( x, 1000, MPI_DOUBLE_PRECISION, src, tag,
     &    MPI_COMM_WORLD, status, ierr )

      end if
```

Distributed Memory: The VECTOR MAX Code

```
  if ( id == 0 ) then                    <-- MASTER receives maximums.

    x_max = -1000.0
    tag = 2

    do i = 1, p - 1

      call MPI_Recv ( x_max_local, 1, MPI_DOUBLE_PRECISION,
&       MPI_ANY_SOURCE, tag, MPI_COMM_WORLD, status, ierr )

      x_max = max ( x_max, x_max_local )

    end do

  else                                   <-- WORKERS send maximums.

    x_max_local = x(1)

    do i = 2, n
      if ( x_max_local .lt. x(i) ) x_max_local = x(i)
    end do

    target = 0
    tag = 2
    call MPI_Send ( x_max_local, 1, MPI_DOUBLE_PRECISION, target,
&     tag, MPI_COMM_WORLD, ierr )

  end if

  return
  end
```

Parallel Programming Concepts

1. Sequential Computing and its Limits
2. Data Dependence
3. Parallel Algorithms
4. Shared Memory Parallel Computing
5. Distributed Memory Parallel Computing
6. **Performance Measurements for Parallel Computing**

What does faster mean? Can you have too many parallel processes?

Performance Measurement

Performance measurements allow us to estimate whether our program is running at maximum speed on a given computer.

They can help us to estimate the running time required if we double the problem size **N**.

They can help us compare algorithms, compilers, computers.

We can try to understand how the problem size **N** and the number of parallel processes **P** interact, so that we can find the "sweet spot", the range of parameters that yield good performance.

Performance Measurement

A common performance measurement is the **MegaFLOPS** rate.

This measurement concentrates entirely on the speed with which a given set of floating-point arithmetic operations is carried out.

A **FLOP** is a floating point operation: addition or multiplication.

We can sometimes estimate the **FLOPs** required for a calcuation.

Matrix multiplication takes $2N^2$,
typical Gauss elimination requires about $\frac{2}{3}N^3$.

Performance Measurement - Time in Seconds

Most computer languages and API's can measure and return a CPU time or wallclock time in seconds:

```
t1 = cputime ( );
x = very_big_calculation ( y );
t2 = cputime ( );
cpu_elapsed = t2 - t1;
```

Performance Measurement

To keep things from getting too big, we typically scale the **FLOPS** by a million.

Now if we divide the work, in **MegaFLOPs**, by the elapsed CPU time, in **seconds**, we get the **MegaFLOPS** rate:

```
MFLOPS = ( FLOPS / 1,000,000 ) / seconds
```

Performance Measurement

A **MegaFLOPS** rate is a reasonable number to use when comparing programs or computers.

Even on a sequential (non-parallel) computation, there can be variations in the computed rate. It's important to run a "big enough" program for a "long enough" time so that the rate settles down.

A **MegaFLOPS** rate is at the heart of the LINPACK benchmark program, which measures the speed at which a 1000x1000 linear system is solved.

Performance Measurement - The LINPACK Benchmark

Table: Sample LINPACK Ratings

Rating	Computer	Language	Comment
108	Apple G5	C	used "rolled" loops
184	Apple G5	C	used "unrolled" loops
221	Apple G5	F77	gfortran compiler
227	Apple G5	Java	
20	Apple G5	MATLAB	using "verbatim" LINPACK
1857	Apple G5	MATLAB	using MATLAB "backslash"

Performance Measurement - The LINPACK Benchmark

Rating	Computer	Site	Processors
12,250,000	2.3 GHz Apple	VT, System X	2,200
13,380,000	Xeon 53xx	"Bank in Germany"	2,640
42,390,000	PowerEdge	Maui	5,200
102,200,000	Cray XT3	"Red Storm", Sandia	26,569
126,900,000	SGI Altix	New Mexico	14,336
167,300,000	BlueGeneP	FZ Juelich	65,536
478,200,000	BlueGeneL	DOE/NNSA/LLNL	212,992
1,026,000,000	Cell+Opteron	Roadrunner, LANL	12,240+6,562

$$1000\,\text{MegaFLOPS} = 1\,\text{GigaFLOPS}$$
$$1000\,\text{GigaFLOPS} = 1\,\text{TeraFLOPS}$$
$$1000\,\text{TeraFLOPS} = 1\,\text{PetaFLOPS} (30\ \text{May}\ 2008)$$

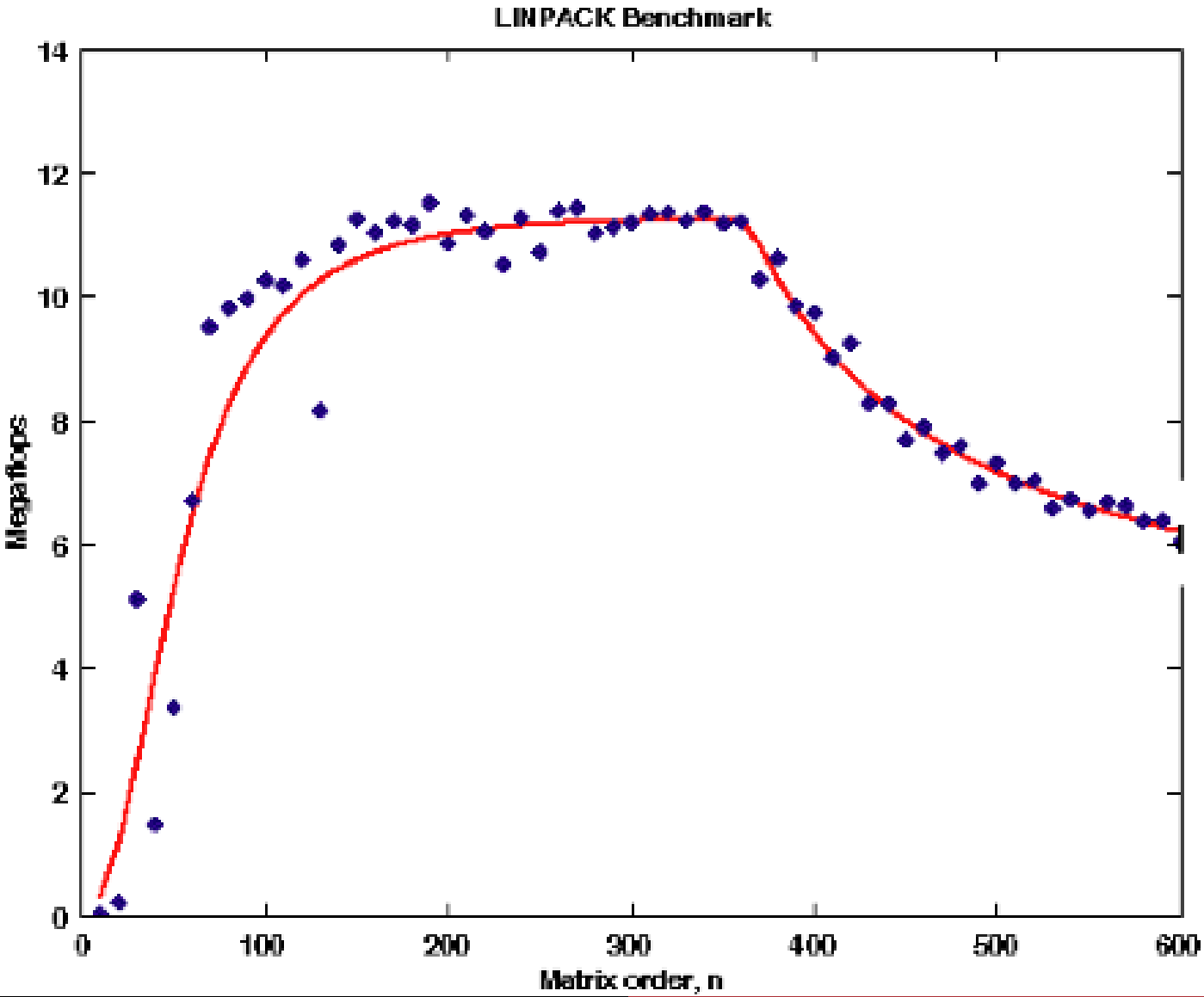
Performance Measurement - The LINPACK Benchmark

Even if you only have your own computer to experiment with, the LINPACK benchmark can sometimes give you some insight into what goes on inside.

Run the program for a series of values of **N**. We often see three phases of behavior:

- a **rising** zone: cache memory and processor are not challenged.
- a **flat** zone, the processor is performing at top efficiency.
- a **decaying** zone, the matrix is so large the cache can't hold enough data to keep the processor running at top speed.

Performance Measurement - The LINPACK Benchmark



Performance Measurement

CPU time is the right thing to measure for sequential computing,

Typical runs occur on a time-shared computer.
Your main cost is the arithmetic calculations.

C/C++:

```
ctime = ( double ) clock ( )
      / ( double ) CLOCKS_PER_SEC;
```

FORTRAN90:

```
call cpu_time ( ctime )
```

Performance Measurement

Wallclock time is what you measure for parallel programming.

A typical run is on a dedicated set of processors.
Cost is number of processors times the elapsed time.

OpenMP (C/C++/FORTRAN):

```
wtime = omp_get_wtime ( );
```

MPI (C/C++/FORTRAN):

```
wtime = MPI_Wtime ( );
```

Performance Measurement - Wallclock Time

Wallclock time forces us to count communication, memory access, and other noncomputational costs.

The total CPU time of a parallel computation is usually significantly higher than for a sequential computation.

We assume (correctly) that individual processors are cheap; our concern is to cut the real time required for an answer.

We can still compute MegaFLOPS ratings, but our timings will be in terms of wall clock time!

Performance Measurement - Problem Size

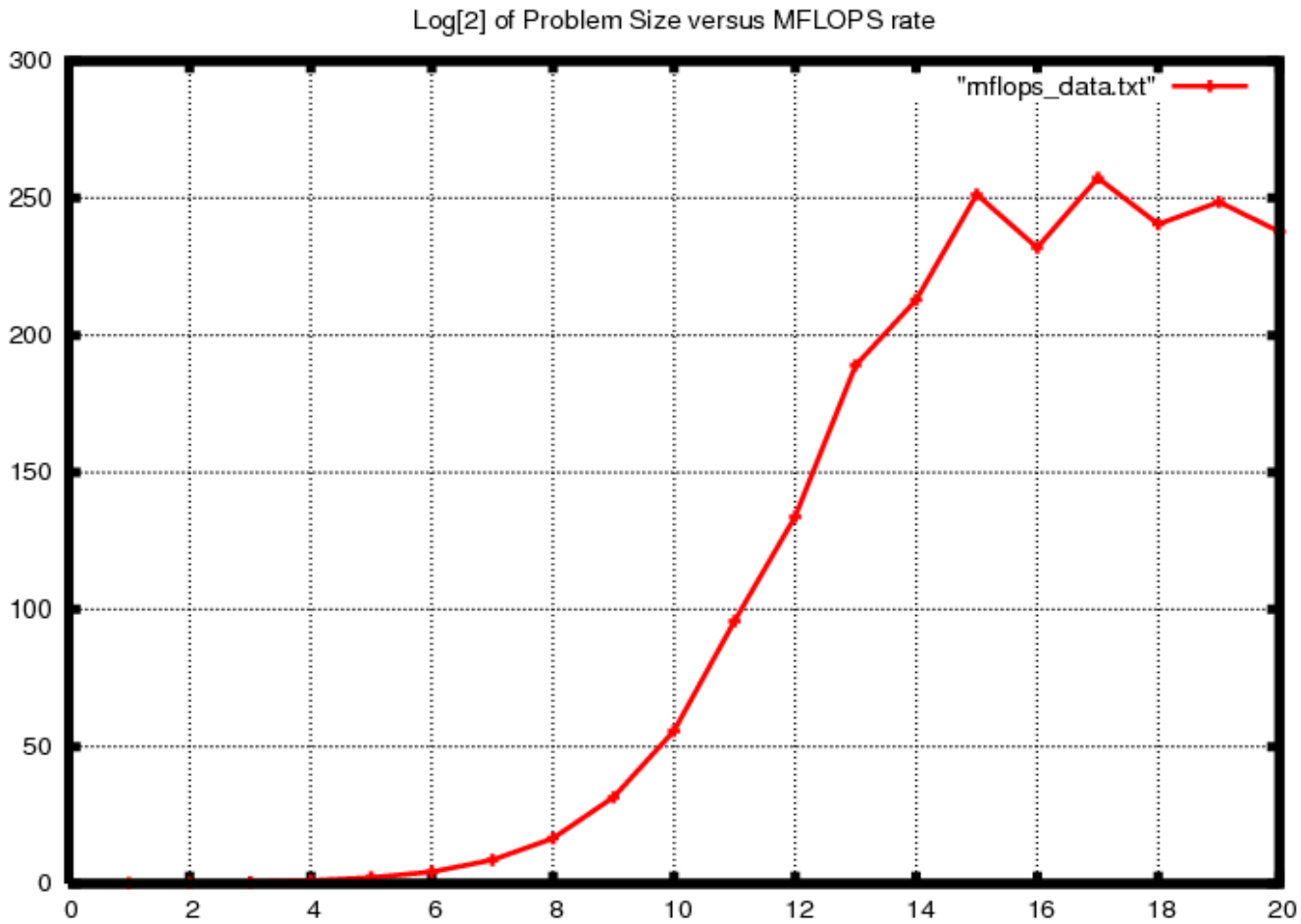
Problem size, symbolized by **N**, affects our performance ratings. Very small and very large problems will do badly, of course. What happens for "reasonable" values of **N**?

Fix the number of processors **P**, and record the solution time **T** for a range of **N** values.

For the Fast Fourier Transform, we know the algorithm, so we know the number of **FLOPS**, and can compute a MegaFLOPS rate.

If we are using the machine well, it should be able to work at roughly a constant rate for a wide range of **N**.

MFLOPS Plot for FFT + OpenMP on 2 Processors



Performance Measurement - Problem Size

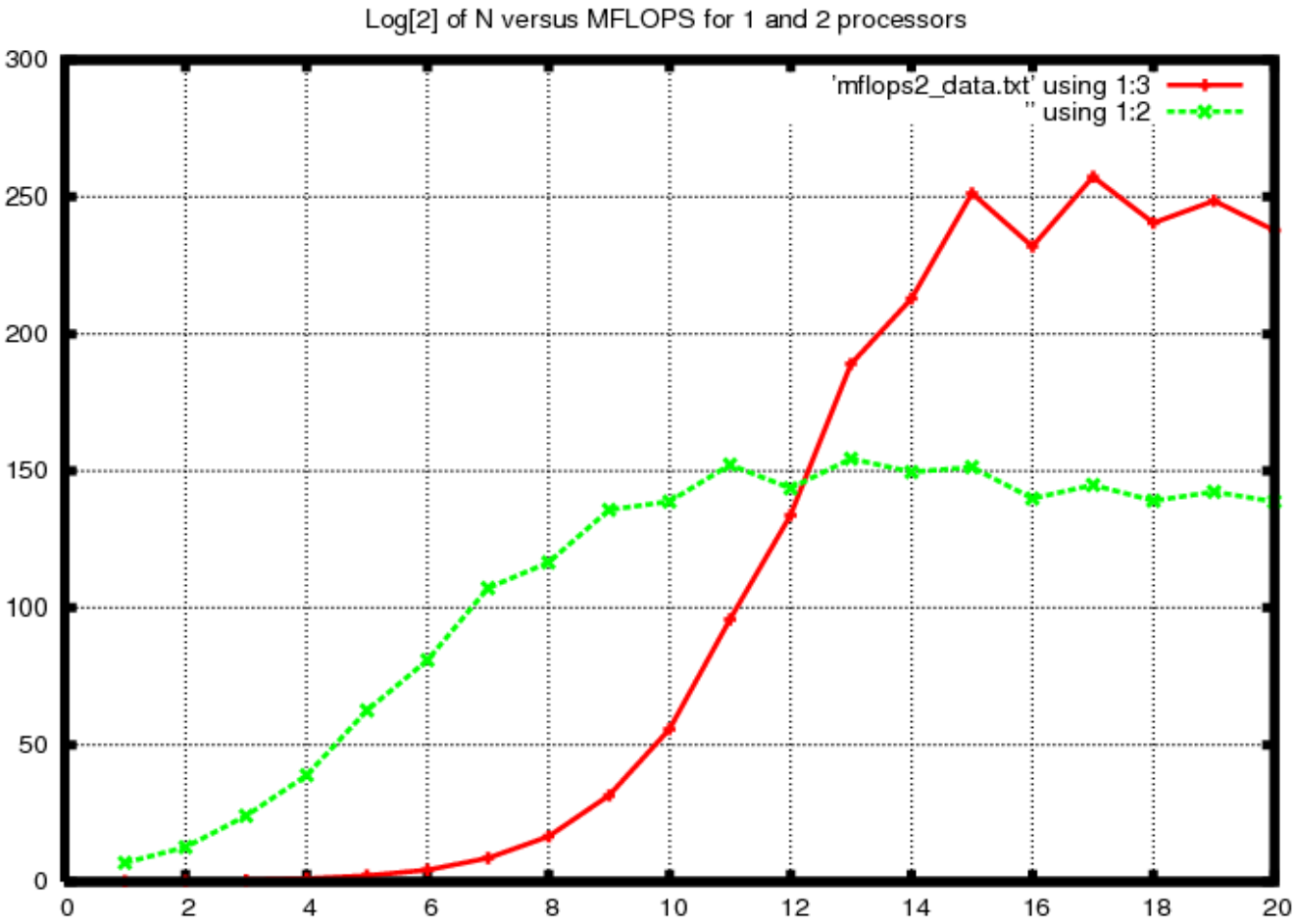
The story is incomplete though! Let's go back (like I said you should do!) and gather the same data for 1 processor.

It's useful to ask ourselves, before we see the plot, what we expect or hope to see. Certainly, we'd like evidence that the parallel code was faster.

But what about the range of values of **N** for which this is true?

Here we will see an example of a "crossover" point. For some values of **N**, running on one processor gives us a better computational rate (**and** a faster execution time!)

MFLOPS Plot for FFT + OpenMP on 1 versus 2 Processors



Performance Measurement - Problem Size

Now, roughly speaking, we can describe the behavior of this FFT algorithm, (at least on 2 processors) as having three ranges, and I will include another which we do not see yet:

- **startup phase**, of very slow performance;
- **surge phase**, in which performance rate increases linearly;
- **plateau phase**, in which performance levels off;
- **saturation phase**, in which performance deteriorates;

Performance Measurement - Problem Size

It should become clear that a parallel program may be better than a sequential program...*eventually* or *over a range of problem sizes* or *when the number of processors is large enough*.

Just from this examination of problem size as an influence on performance, we can see evidence that parallel programming comes with a *startup cost*. This cost may be excessive for small problem size or number of processors.

Performance Measurement

While a MegaFLOPS rate can be very illuminating, it's only computable if the FLOPs can be determined.

That assumes the entire computation is basically one simple algorithm (Gauss elimination, FFT, etc).

This is NOT true for most scientific programs of interest.

Program run time is very accessible. We can learn by studying its dependence on the number of processors, and on the problem size.

Performance Measurement - Speedup

Speedup measures the work rate as we add processors.

The work **W** depends on the problem size **N**.

For a fixed **N**, we solve the problem for an increasing series of processors **P**, and record the time **T**.

For a "perfect speedup", we would hope that

$$T = W/P$$

Doubling the processors would halve the time, and we could drive **T** to zero if we had enough processors.

Performance Measurement - Speedup

To define a formula for speedup **S**, we time the problem solution using 1 processor, and set that to be **T(1)**.

Then if we repeat the computation using **P** processors, the speedup function is defined as

$$S(P) = T(1)/T(P)$$

In a perfect world, **S(P)** would equal **P**.
Tracking **S(P)** indicates how well our program parallelizes.

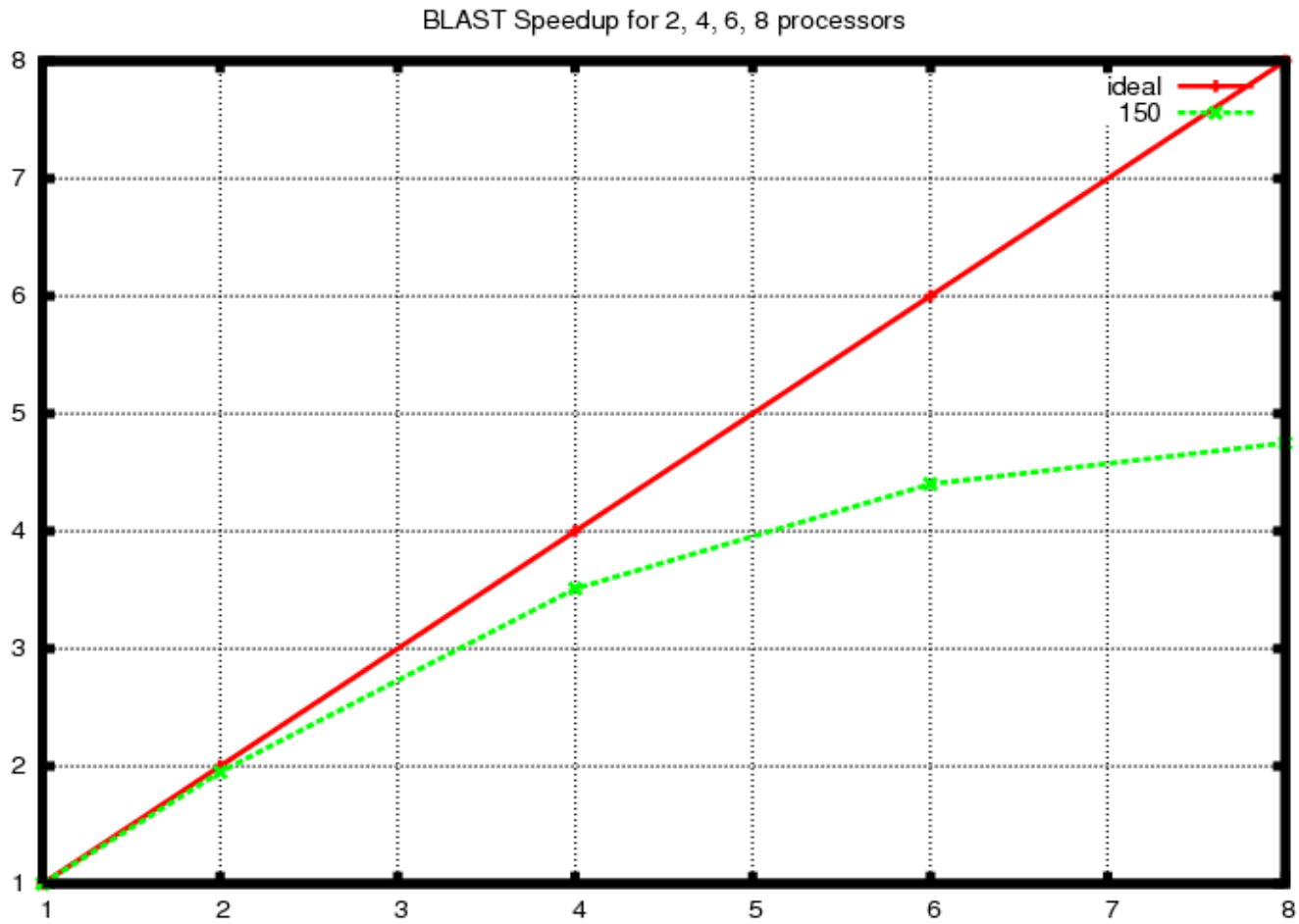
Performance Measurement - Number of Processors

When we plot the speedup function **P** versus **S(P)**, the first data point should be (1,1). This simply means that we are normalizing the data so that the computational rate on one processor is set to 1.

When we run the program on two processors, our time probably won't be half of **T(1)**, but maybe it will be 0.55 times it. We plot the point (2, 1/0.55) = (2, 1.818).

The ideal speedup behavior is a diagonal line, but we will see the actual behavior is a curve that "strives" for the diagonal line at first, and then "gets tired" and flattens out (and will actually decrease if we go out too far!)

Performance Data for a version of BLAST



Performance Measurement - Number of Processors

When you plot data this way, there is a nice interpretation of the scale. In particular, note that when we use 6 processors, our speedup is about 4.5. This means that we when use 6 processors, the program speeds up as though we had 4.5 processors (assuming a perfect speedup).

Some people define the **parallel effficiency** as the ratio of the effective number of processors divided by the actual number. For this case, at 6 processors our efficiency would be 0.75. (It's actually the slope of the speedup graph).

You can also see the curve flattening out as the number of processors increases.

Performance Measurement - Number of Processors

The speedup function really depends on the problem size as well as the number of processors, so we could write **S(P,N)**.

If we fix **P**, and consider a sequence of increasing sizes **N**, we often see the familiar rise, plateau and deterioration behavior.

But it is often the case that the plateau, the range of problem sizes with good performance, moves and widens as we increase **P**.

For a fixed **P**, bigger problems run better.
For a fixed **N**, adding processors reduces time, but also efficiency.

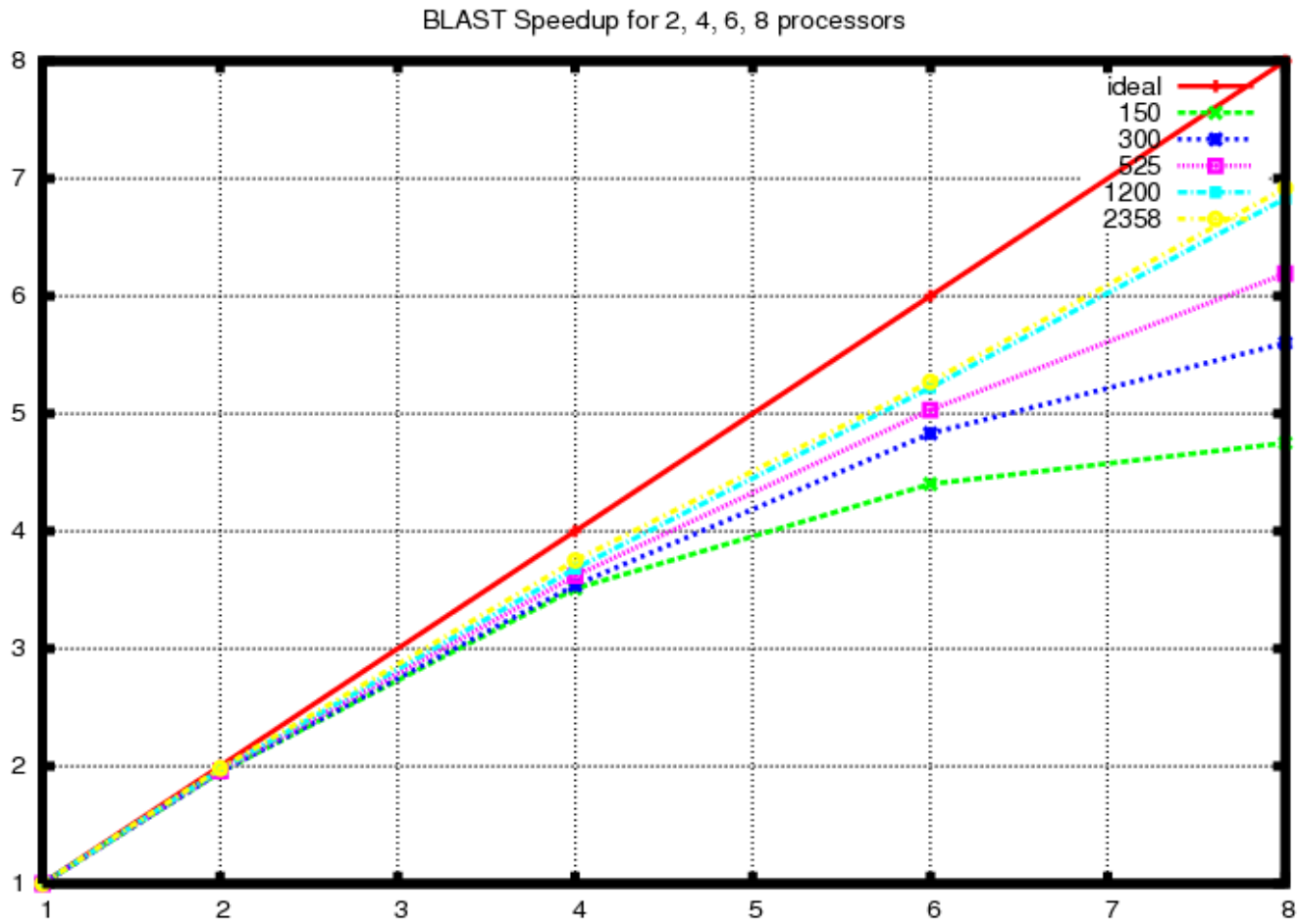
Performance Measurement - Number of Processors

We can try to display the relationship between **P** and **N** on one plot.

The problem size for the BLAST program is measured in the length of the protein sequence being analyzed, so we can't pick **N** arbitrarily.

Let's go back and do timings for sequences of 5 different sizes, and display a series of **S(P)** curves for different **N**.

Performance Data for a version of BLAST



Performance Measurement - Applying for Computer Time

To get time on a parallel clusters, you must justify your request with performance data.

You should prepare speedup plots for a few typical problem sizes **N** and processors from 1, 2, 4, 8, ... to **P**.

If you still get about 30% speedup using **P** processors, your request is reasonable.

In other words, if a 100 processor machine would let you run 30 times faster, you're got a good case.

Parallel Programming Concepts

1. Sequential Computing and its Limits
2. Data Dependence
3. Parallel Algorithms
4. Shared Memory Parallel Computing
5. Distributed Memory Parallel Computing
6. Performance Measurements for Parallel Computing
7. **Auto-Parallelization**

A quick and easy way to try parallel programming on your code.

Auto-Parallelization

Auto-parallelization is a feature, available with some compilers, which attempts to identify portions of the source code which can be executed in parallel.

This is an easy way to experiment with the potential for parallel execution.

This feature is generally only available for use on *shared memory* machines - which includes laptops with dual cores, the Virginia Tech SGI systems.

The executable code must be run in the appropriate parallel environment (that is, you must set the OpenMP environment variable that specifies the number of threads to use).

Auto-Parallelization

Auto-parallelization is *cautious*: its goal is only to make changes that it can guarantee will execute the same on multiple processors.

Auto-parallelization is *limited*: it will miss some chances to parallelize.

Auto-parallelization is *not perfect*: it may be fooled into parallelizing some code when it is not safe to do so.

So code that is auto-parallelized should be **checked for accuracy** as well as **timed for speedup**.

Auto-Parallelization - GCC Compiler Flags

GCC/G++/GFORTRAN compilers:

```
gcc -ftree-vectorize -O2 myprog.c
```

Add the switch -**maltivec** on PowerPC hardware.

Note that -**ftree-vectorize** is one word!

These commands invoke **auto-vectorization**, which is less ambitious. GCC is currently developing auto-parallelization features, but they are not widely used yet.

Auto-Parallelization - IBM Compiler Flags

IBM FORTRAN compiler:

```
xlf_r -O3 -qsmp=auto -qreport=smplist myprog.f
```

IBM C compiler:

```
xlc_r -O3 -qsmp=auto -qreport=smplist myprog.c
```

The optional *-qreport=smplist* switch reports on why some loops could not be parallelized.

There is also a *-qsource* switch which produces a compiler listing.

Auto-Parallelization - Intel Compiler Flags

Intel FORTRAN compiler:

```
ifort -fpp -parallel myprog.f
```

Intel C compiler (icpc is similar):

```
icc -parallel myprog.c
```

The optional *-par_report3* switch reports on why some loops could not be parallelized.

Auto-Parallelization - SGI Compiler Flags

SGI FORTRAN compilers f77/f90:

```
f77 -O3 -apo myprog.f
```

SGI C compilers cc/c++:

```
cc -O3 -apo myprog.c
```

Replace the *-apo* switch by the *-apolist* switch in order to get a listing that explains why some loops were not parallelized.

Auto-Parallelization - Sun Compiler Flags

Sun FORTRAN compilers f77/f90/f95:

```
f77 -fast -autopar -parallel -loopinfo myprog.f
```

Sun C compilers:

```
cc -xautopar -xparallel -xloopinfo myprog.c
```

The optional *loopinfo* switch reports on why some loops could not be parallelized.

Auto-Parallelization - Execution

To run the program, first set the OpenMP environment variable to indicate how many threads you want (perhaps 2 or 4):

Bourne, Korn and Bash shell users:

```
export OMP_NUM_THREADS=4
```

C and T shell users:

```
setenv OMP_NUM_THREADS 4
```

Then run your program with the usual **./a.out** command.

Auto-Parallelization - Obstacles to good results

- loops that contain output statements
- exiting a loop early
- **while** loops
- loops that call functions
- loops with very little work
- loops with low iteration count - **N** is small
- loops that compute scalar functions of vector data: maximum, dot product, norm, integral estimates
- loops with complicated array indexing - possible data dependence
- loops with actual data dependence

Auto-Parallelization

The code produced by the auto-parallelizer may run more slowly, or even incorrectly.

However, it can be a very useful place from which to begin parallelization efforts.

Some compilers offer a listing that shows why some loops were not parallelized. Working from this listing, it may be possible to modify loops so that the auto-parallelizer can handle them.

Moreover, the auto-parallizer may even produce a revised version of the source code, including parallelization directives. This code may be very useful as a starting point for further parallelization "by hand."

Conclusion - Don't Panic!

A simple model of the two kinds of parallel programming:

- **shared memory**, more than one processor can see the problem and help out; sometimes processors can get in the way of each other
- **distributed memory**, the problem is divided up in some way; processors work on their part in isolation, and send messages to communicate

In the next few days, we'll **implement** these ideas to make practical parallel programs.

Conclusion - Don't Panic