POLITECNICO DI MILANO

Design of Parallel Programs

Algoritmi e Calcolo Parallelo

Daniele Loiacono

References

- The material in this set of slide is taken from two tutorials by Blaise Barney from the Lawrence Livermore National Laboratory and from slides of prof. Lanzi (Informatica B, A.A. 2009/2010)
- Introduction to Parallel Computing
  Blaise Barney, Lawrence Livermore National Laboratory
  https://computing.llnl.gov/tutorials/parallel_comp/
- Also available as Dr.Dobb’s “Go Parallel”
  Introduction to Parallel Computing: Part 2
  Blaise Barney, Lawrence Livermore National Laboratory

Parallel program design

Steps to Parallelization

Understand the Problem and the Program

Partitioning
(domain vs functional decomposition)

Communication
(cost, latency, bandwidth, visibility, synchronization, etc.)

Data Dependencies

Understand the Problem and the Program

- □ Problem A: Calculate the potential energy for each of several thousand independent conformations of a molecule. When done, find the minimum energy conformation.

- □ Problem B: 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)

Which one can be parallelized?

Why?

Understand the Problem and the Program

- Identify the program's hotspots
  - Know where most of the real work is being done
  - Most programs accomplish most of their work in a few places. (profilers and performance analysis tools)
  - Focus on parallelizing the hotspots
- Identify bottlenecks in the program
  - Are there areas that are disproportionately slow, or cause parallelizable work to halt or be deferred? (I/O)
  - May be possible to restructure the program or use a different algorithm to reduce or eliminate unnecessary slow areas
- Identify inhibitors to parallelism
  - Data dependence, …
- Investigate other algorithms if possible

Decomposition or Partitioning

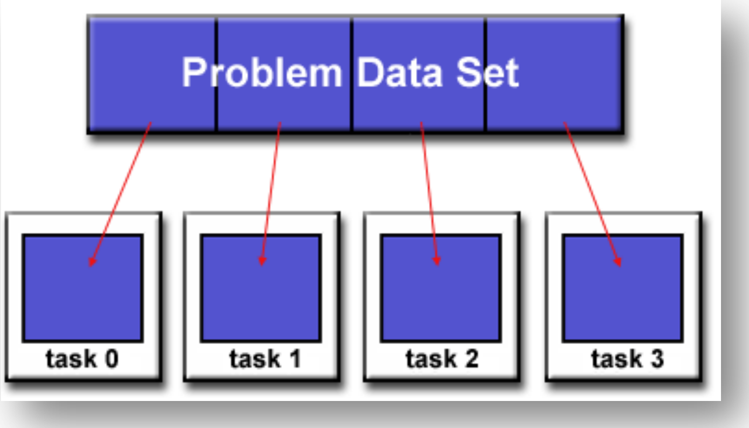
- One of the first steps in designing a parallel program

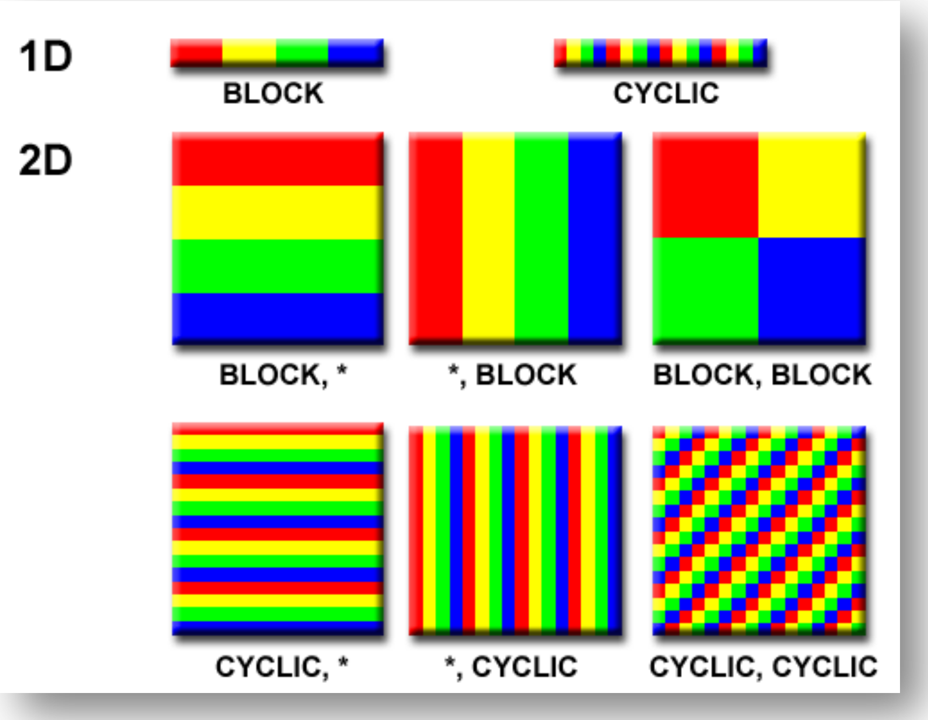
- Break the problem into discrete "chunks" of work that can be distributed to multiple tasks. This is known as decomposition or partitioning.

- Two ways to partition computation among parallel tasks
  - Domain decomposition
  - Functional decomposition

Domain Decomposition
(Focus on the data)

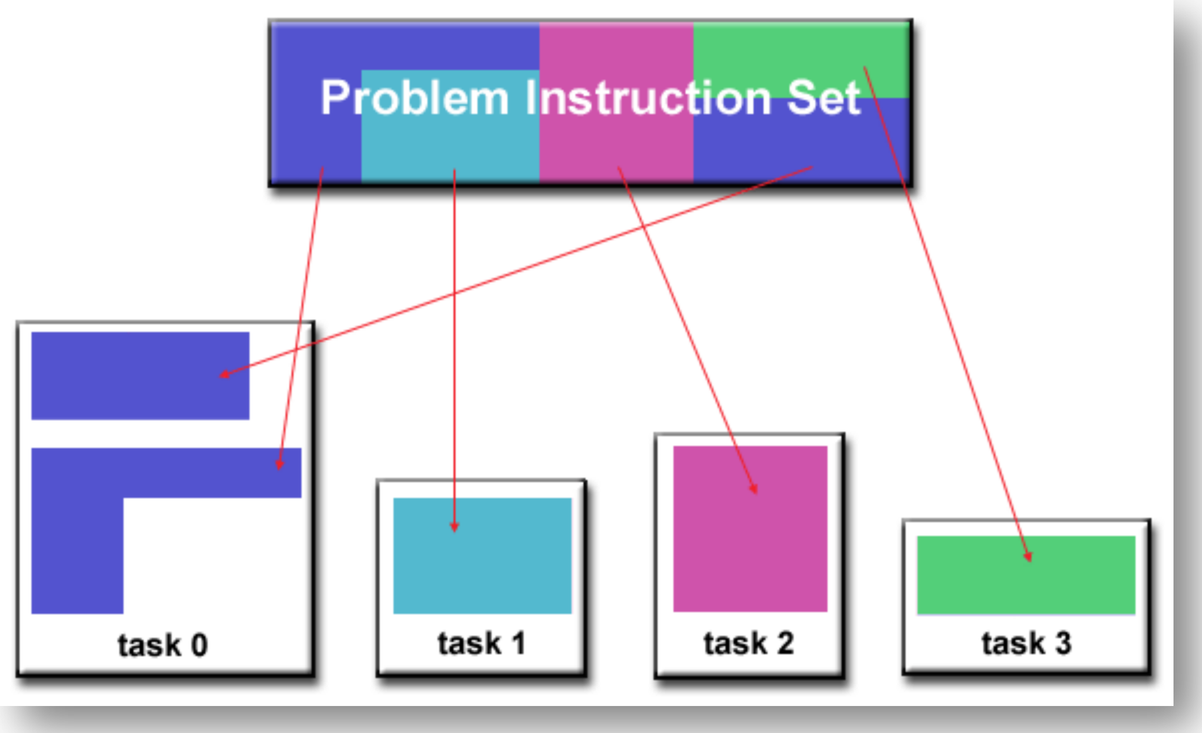
- The data associated with a problem is decomposed
- Each parallel task then works on a portion of the data





Functional Decomposition (Focus on the computation)

- The focus is on the computation that is to be performed rather than on the data manipulated by the computation
- The problem is decomposed according to the work that must be done. Each task then performs a portion of the overall work.
- Functional decomposition lends itself well to problems that can be split into different tasks.



Designing Parallel Programs: Communication

- Communications between tasks depends upon the problem

- No Need for communications
  - Problems that can be decomposed and executed in parallel with virtually no need for tasks to share data.
  - Example: image processing where computation is local
  - Often called embarrassingly parallel because they are so straight-forward

- Need for communication
  - Most parallel applications require tasks to share data
  - 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.

What Factors to Consider?

Cost of Communications

- Inter-task communication always implies overhead
- Resources are used to package/transmit data instead of computation
- 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 or gigabytes/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

What Factors to Consider?

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

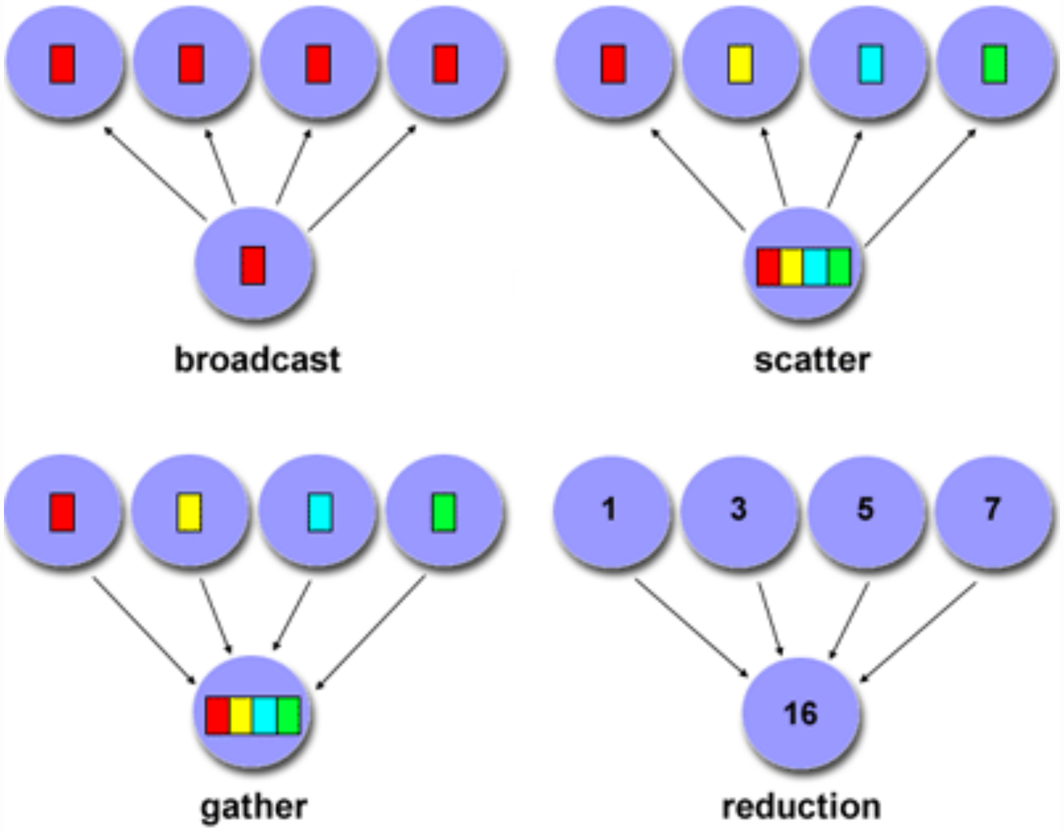
- ❑ Synchronous vs. asynchronous communications
  - Synchronous communications require handshaking between tasks that are sharing data (explicitly encoded or transparent to the programmer)
  - Synchronous communications are blocking since other work must wait until the communications have completed.
  - Asynchronous communications allow tasks to transfer data independently from one another
  - Asynchronous communications are non-blocking 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

What Factors to Consider?
Scope of the Communication

- Knowing which tasks must communicate with each other is critical during the design stage of a parallel code

- Point-to-point - involves two tasks with one task acting as the sender/ producer of data, and the other acting as the receiver/consumer.

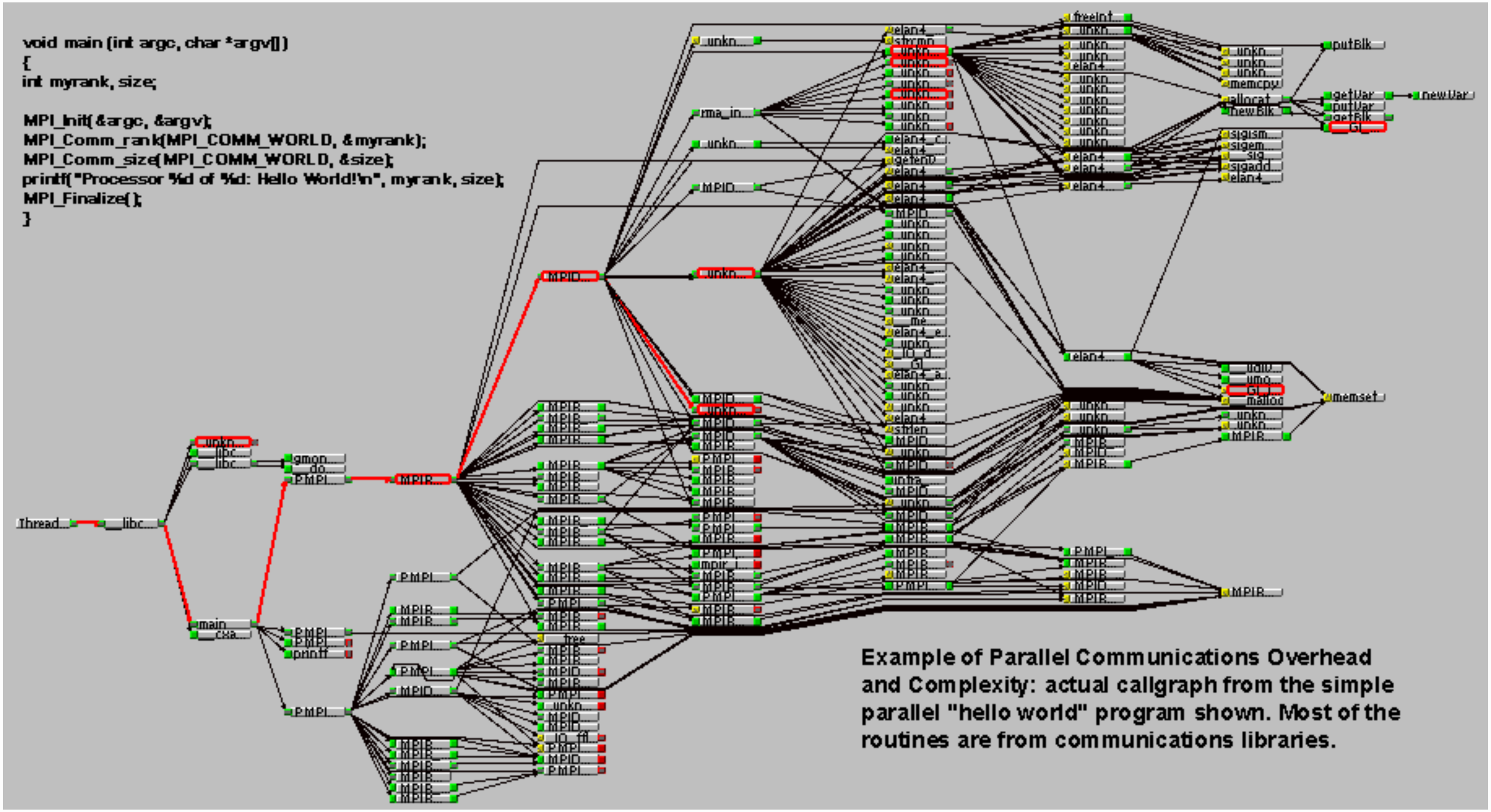
- Collective - involves data sharing between more than two tasks, which are often specified as being members in a common group, or collective. Some common variations (there are more)



What Factors to Consider?

- **Efficiency of communications**
  - The programmer often has a choice with regard to factors that can affect communications performance.
  - 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?

What Factors to Consider?
Overhead and Complexity



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

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

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

Design of Parallel Program: Synchronization

- **Barrier**
  - Usually implies that all tasks are involved
  - Each task performs its work until it reaches the barrier, then it "blocks"
  - When the last task reaches the barrier, all tasks are synchronized
  - What happens from here varies (serial section, release the tasks, etc.)
- **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 the lock/semaphore/flag
  - The first task to acquire the lock "sets" it. This task can then safely (serially) access the protected data or code.
  - Other tasks can attempt to acquire the lock but must wait until the task that owns the lock releases it.
  - Can be blocking or non-blocking
- **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.

Designing Parallel Programs
Data Dependencies

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

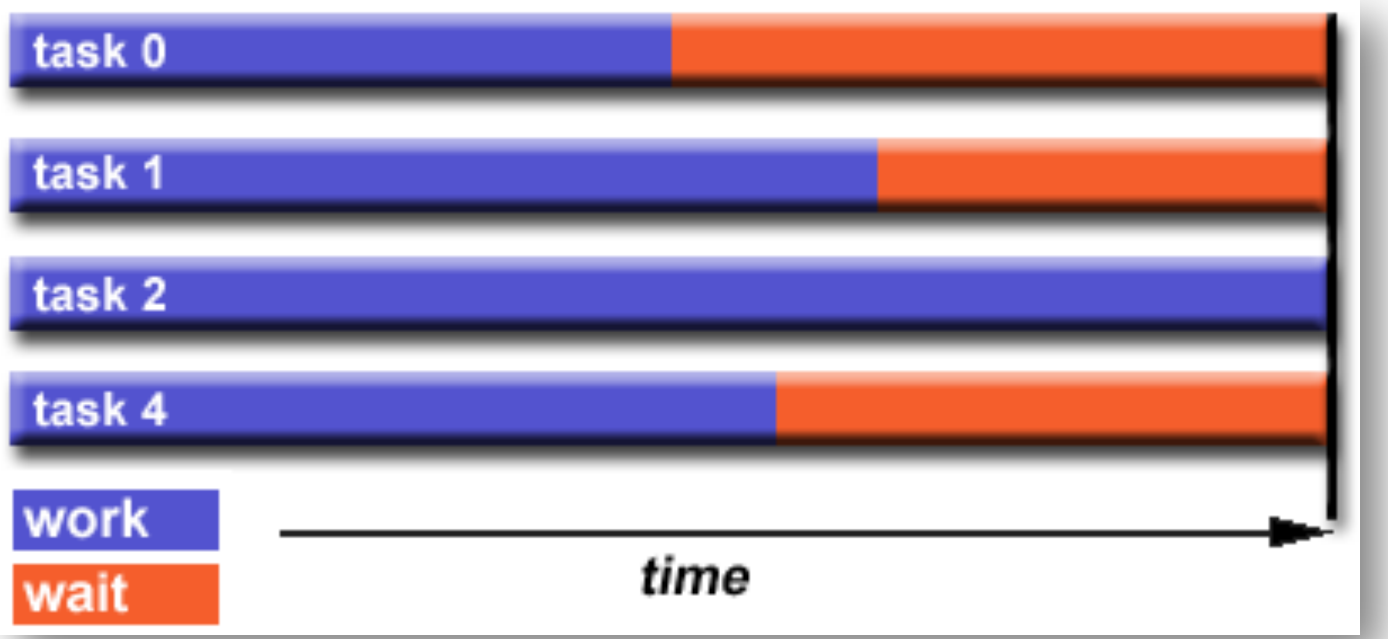
```
for (int i=1; i<500; i++)
    a[i]=a[i-1] * 2;
```

  - The value of `a[i-1]` must be computed before the value of `a[i-1]`, therefore `a[i]` exhibits a data dependency on A(J-1). Parallelism is inhibited.

Load balancing…

Designing Parallel Programs: Load Balancing

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



How to Achieve Load Balance? Equally Partition the Work

- For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks

- For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.

- If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

How to Achieve Load Balance? Use Dynamic Work Assignment

- Certain classes of problems result in load imbalances even if data is evenly distributed among tasks:
  - Sparse arrays - some tasks will have actual data to work on while others have mostly "zeros".
  - Adaptive grid methods - some tasks may need to refine their mesh while others don't.
  - N-body simulations - where some particles may migrate to/from their original task domain to another task's; where the particles owned by some tasks require more work than those owned by other tasks.

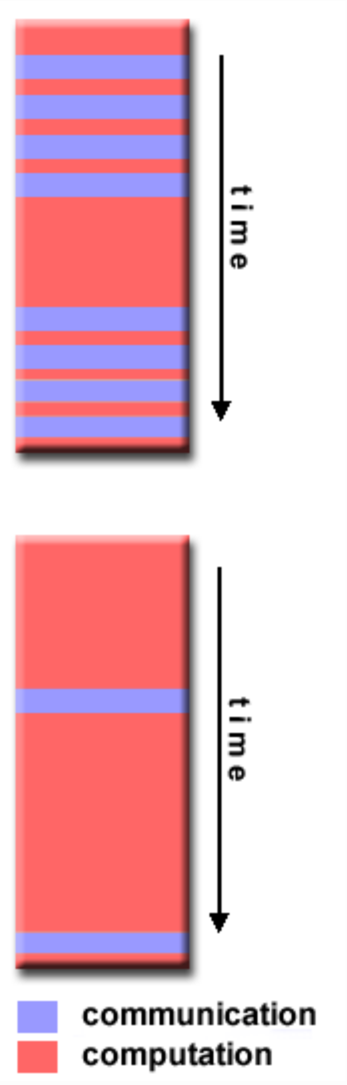
- When the amount of work each task will perform is intentionally variable, or is unable to be predicted, it may be helpful to use a scheduler - task pool approach. As each task finishes its work, it queues to get a new piece of work.

- It may become necessary to design an algorithm which detects and handles load imbalances as they occur dynamically within the code.

Granularity

Designing Parallel Programs Granularity

- ❑ **Computation/Communication Ratio**
  - Granularity is a qualitative measure of the ratio of computation to communication.
  - Periods of computation are typically separated from periods of communication by synchronization events.



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

Designing Parallel Programs
Granularity

- Coarse-grain Parallelism
  - Relatively large amounts of computational work are done between communication/synchronization events
  - High computation to communication ratio
  - Implies more opportunity for performance increase
  - Harder to load balance efficiently

- Which is Best?
  - The most efficient granularity is dependent on the algorithm and the hardware environment in which it runs.
  - In most cases the overhead associated with communications and synchronization is high relative to execution speed so it is advantageous to have coarse granularity.
  - Fine-grain parallelism can help reduce overheads due to load imbalance.

Input/output…

Designing Parallel Programs: Input/Output

- The Bad News
  - I/O operations are generally regarded as inhibitors to parallelism
  - Parallel I/O systems may be immature or not available
  - In an environment where all tasks see the same file space, write operations can result in file overwriting
  - Read operations can be affected by the file server's ability to handle multiple read requests at the same time
  - I/O that must be conducted over the network (NFS, non-local) can cause severe bottlenecks and even crash file servers
- The Good News
  - Parallel file systems are available
  - Examples: General Parallel File System for AIX by IBM; Lustre: for Linux clusters by SUN Microsystems, PVFS/PVFS2: Parallel Virtual File System for Linux clusters; etc.
  - The parallel I/O programming interface specification for MPI has been available since 1996 as part of MPI-2. Vendor and "free" implementations are now commonly available.

Some Options for Managing I/O

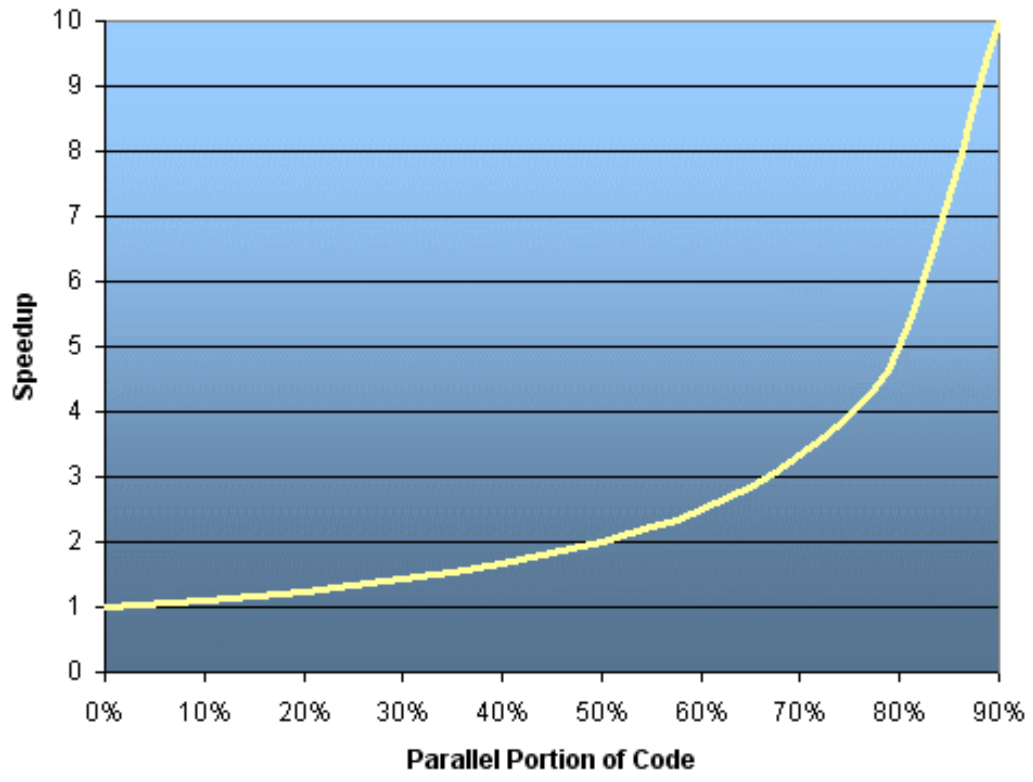
- If you have access to a parallel file system, investigate using it
- Rule #1: Reduce overall I/O as much as possible
- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks. For example, Task 1 could read an input file and then communicate required data to other tasks. Likewise, Task 1 could perform write operation after receiving required data from all other tasks.
- For distributed memory systems with shared filespace, perform I/O in local, non-shared filespace. For example, each processor may have /tmp filespace which can used. This is usually much more efficient than performing I/O over the network to one's home directory.
- Create unique filenames for each task's input/output file(s)

Limits and costs…

Designing Parallel Programs: Speedup

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

  speedup = 1/(1-P)



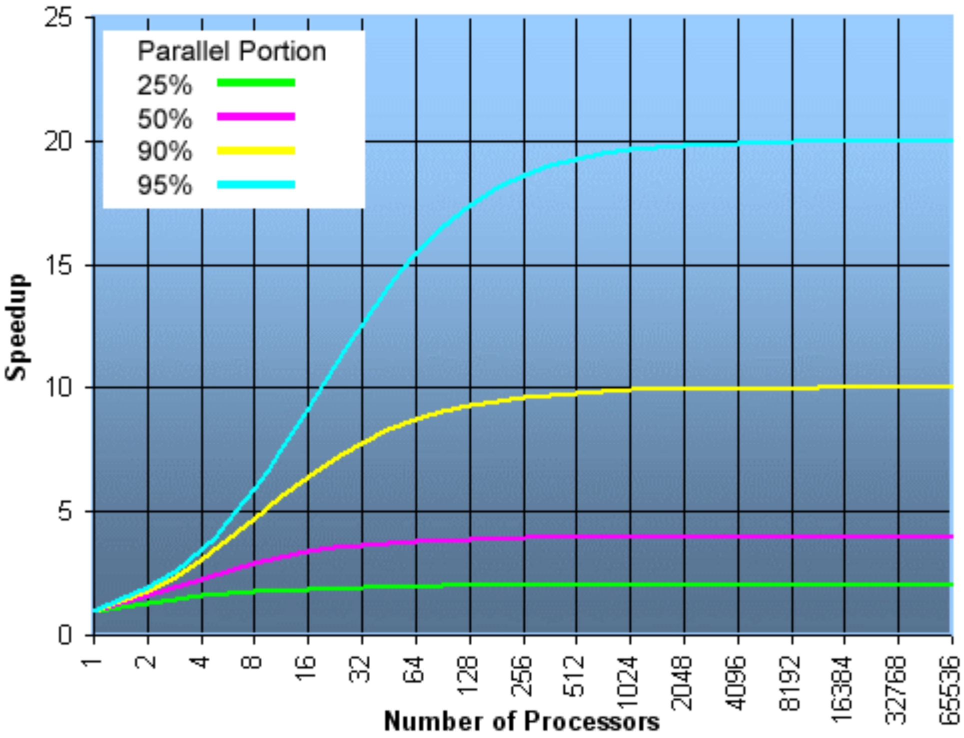
Designing Parallel Programs: Speedup

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

speedup = 1/(P/N+S)

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

- It soon becomes obvious that there are limits to the scalability of parallelism.



Designing Parallel Programs: Speedup and Scalability

- However, certain problems demonstrate increased performance by increasing the problem size. For example:

2D Grid Calculations	85 seconds	85%
Serial fraction	15 seconds	15%

- We can increase the problem size by doubling the grid dimensions and halving the time step. This results in four times the number of grid points and twice the number of time steps. The timings then look like:

2D Grid Calculations	680 seconds	97.84%
Serial fraction	15 seconds	2.16%

- Problems that increase the percentage of parallel time with their size are more scalable than problems with a fixed percentage of parallel time.

Designing Parallel Programs: Complexity

- Parallel applications are much more complex than corresponding serial applications, perhaps an order of magnitude.
- Not only do you have multiple instruction streams executing at the same time, but you also have data flowing between them.
- The costs of complexity are measured in programmer time in virtually every aspect of the software development cycle: design, coding, debugging, tuning and maintenance.
- Adhering to "good" software development practices is essential when working with parallel applications - especially if somebody besides you will have to work with the software.

Designing Parallel Programs: Portability

- Standardization in several APIs, such as MPI, POSIX threads, HPF and OpenMP, has reduced the portability issues of the years past.
- All of the usual portability issues associated with serial programs apply to parallel programs. For example, if you use vendor "enhancements" to Fortran, C or C++, portability will be a problem
- Even though standards exist for several APIs, implementations will differ in a number of details, sometimes to the point of requiring code modifications in order to effect portability
- Operating systems can play a key role in code portability issues
- Hardware architectures are characteristically highly variable and can affect portability

Designing Parallel Programs: Resource Requirements

- Parallel programming aims at decreasing execution wall clock time, but it achieves this by using more CPUs

- For example, a parallel code that runs in 1 hour on 8 processors actually uses 8 hours of CPU time.

- The amount of memory required can be greater for parallel codes, due to the need to replicate data and for overheads associated with parallel support libraries and subsystems.

- For short running parallel programs, there can actually be a decrease in performance compared to a similar serial implementation.

- The overhead costs associated with setting up the parallel environment, task creation, communications and task termination can comprise a significant portion of the total execution time for short runs.

Designing Parallel Programs: Scalability

- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more machines is rarely the answer.

- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Most parallel solutions demonstrate this characteristic at some point.

- Hardware factors play a significant role in scalability.

- Parallel support libraries and subsystems software can limit scalability independent of your application.

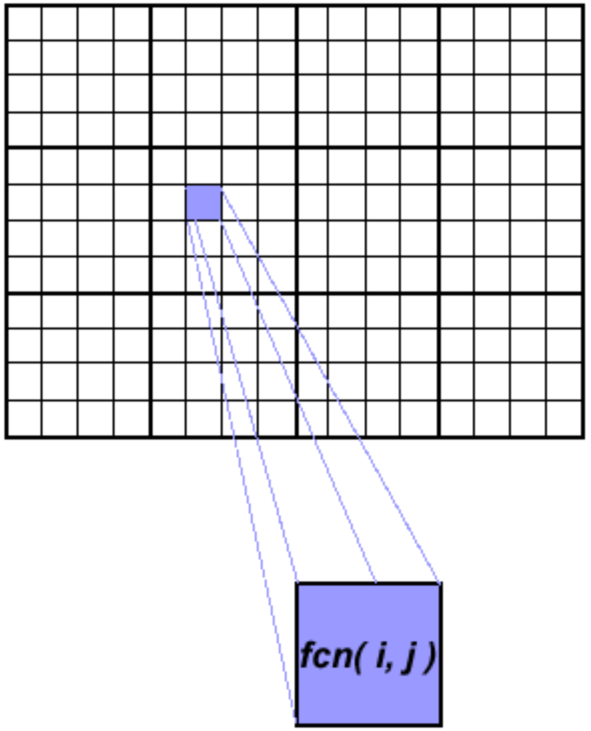
Examples…

Array Processing

- ❑ Problem: calculations on 2-dimensional array elements, with the computation on each array element being independent from other array elements.

- ❑ Serial Solution

```
for(i=0; i<n; i++)
    for(j=0; j<n; j++)
          a[i][j] = fcn(i,j)
```



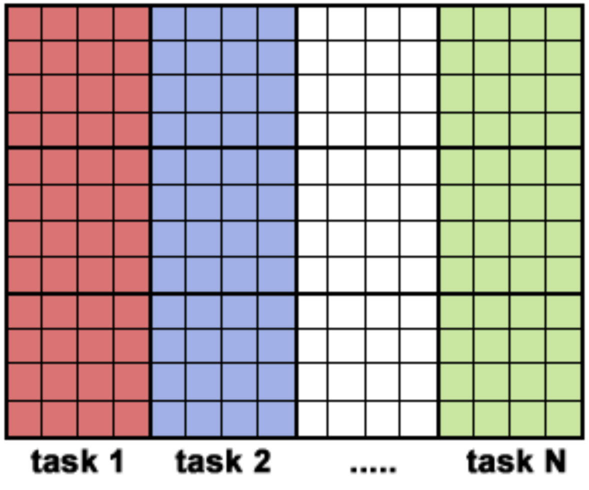
- ❑ The calculation of elements is independent of one another

- ❑ Leads to an embarrassingly parallel situation.

Array Processing: Parallel Solution

- 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.

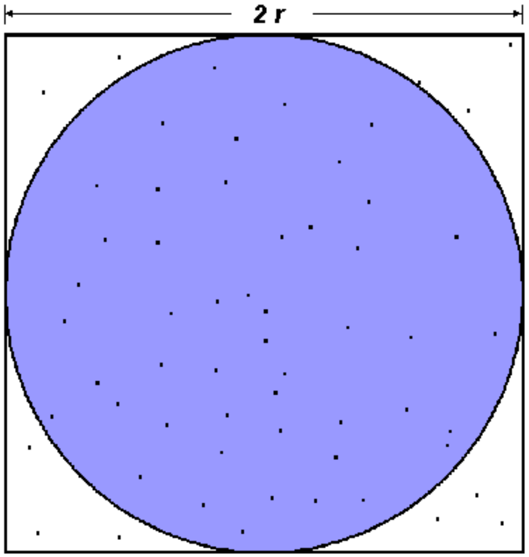


- After the array is distributed, each task executes the portion of the loop corresponding to the data it owns. For example,

```
for(i=start_block; i<end_block; i++)
    for(j=0; j<n; j++)
          a[i][j] = fcn(i,j)
```

PI Computation

- Method of approximating PI
  - Inscribe a circle in a square
  - Randomly generate points in the square
  - Compute the number of points in the square that are also in the circle
  - Let x be the number of points in the circle divided by the number of points in the square
  - PI ~ 4 x
- The more points generated, the better the approximation



$$A_S = (2r)^2 = 4r^2$$
$$A_C = \pi r^2$$
$$\pi = 4 \times \frac{A_C}{A_S}$$

```
npoints = 10000; circle_count = 0;
for(j=1, j<npoints; j++) {
     xcoordinate = random1;
     ycoordinate = random2;
     if (xcoordinate, ycoordinate) inside circle
             then circle_count = circle_count + 1
}
PI = 4.0*circle_count/npoints
```

PI Computation
Parallel Solution

- Embarrassingly parallel solution
  - Computationally intensive
  - Minimal communication
  - Minimal I/O

- Parallelization: break the loop into portions that can be executed by the tasks.

- For the task of approximating PI:
  - Each task executes its portion of the loop
  - Each task can do its work without requiring any information from the other tasks (no data dependencies).
  - Uses the SPMD model. One task acts as master and collects the results.

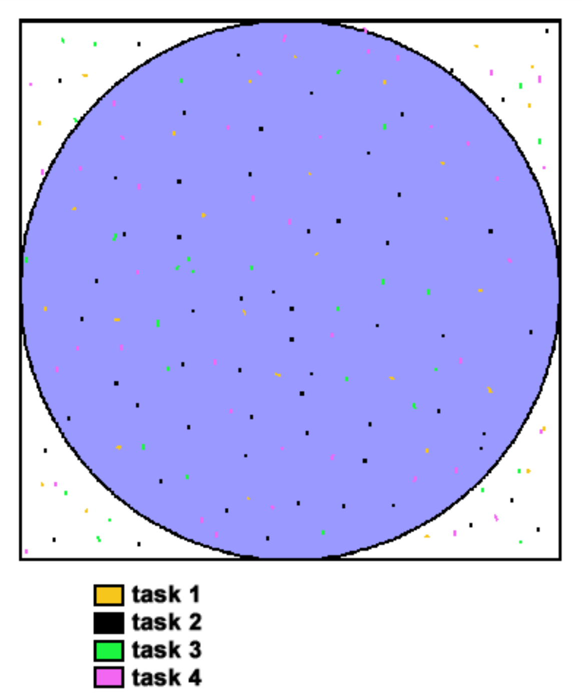
PI Computation
Parallel Solution

```
npoints = 10000; circle_count = 0;
p = number of tasks; num = npoints/p;

find out if I am MASTER or WORKER

for(j=1, j<num; j++) {
    x = random();
    y = random();
    if ((x, y) inside circle)
        circle_count = circle_count + 1
}

if I am MASTER
    receive from WORKERS their circle_counts
    compute PI (use MASTER and WORKER calculations)
else if I am WORKER {
   send to MASTER circle_count
}
```


task 1
task 2
task 3
task 4