Parallel & Distibuted Computing: Lecture 4

from Blaise N. Barney, HPC Training Materials, by kind permission of Lawrence Livermore National Laboratory's Computational Training Center

March 15, 2017

Concepts and Terminology

1. General concepts
2. Limits and Costs of Parallel Programming
3. Sequential implementation of domain integration of polynomials

General concepts

von Neumann Computer Architecture

- Named after the Hungarian mathematician/genius John von Neumann who first authored the general requirements for an electronic computer in his 1945 papers.
- Also known as "stored-program computer" - both program instructions and data are kept in electronic memory. Differs from earlier computers which were programmed through "hard wiring".
- Since then, virtually all computers have followed this basic design

Figure 1: John von Neumann circa 1940s (Source: LANL archives)

von Neumann Computer Architecture

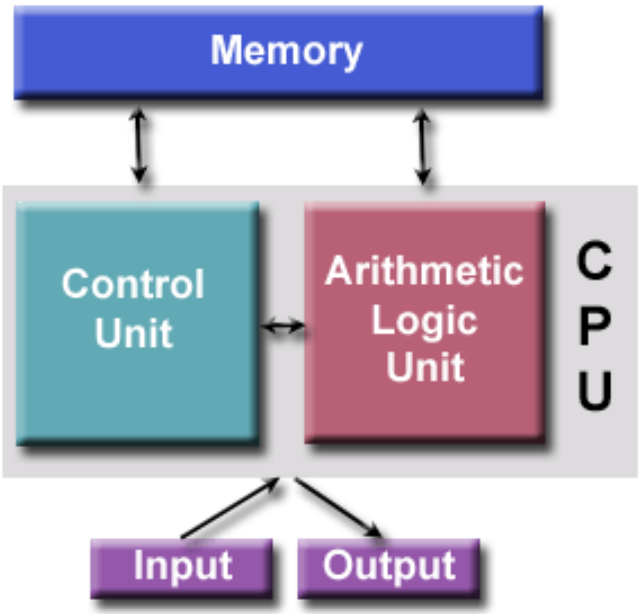


Figure 2: von Neumann Architecture

- Four main components:
  - Memory
  - Control Unit
  - Arithmetic Logic Unit
  - Input/Output
- Read/write, random access memory to store both program instructions and data
  - Program instructions are coded data which tell the computer to do something
  - Data is information to be used by the program
- Control unit fetches instructions/data from memory, decodes the instructions and then sequentially coordinates operations to accomplish the programmed task.
- Arithmetic Unit performs basic arithmetic operations
- Input/Output is the interface to the human operator

Flynn's Classical Taxonomy

- There are different ways to classify parallel computers. Examples available HERE.
- One of the more widely used classifications, in use since 1966, is called Flynn's Taxonomy.
- Flynn's taxonomy distinguishes multi-processor computer architectures according to how they can be classified along the two independent dimensions of Instruction Stream and Data Stream.
- Each dimension can have only two possible states: Single or Multiple.

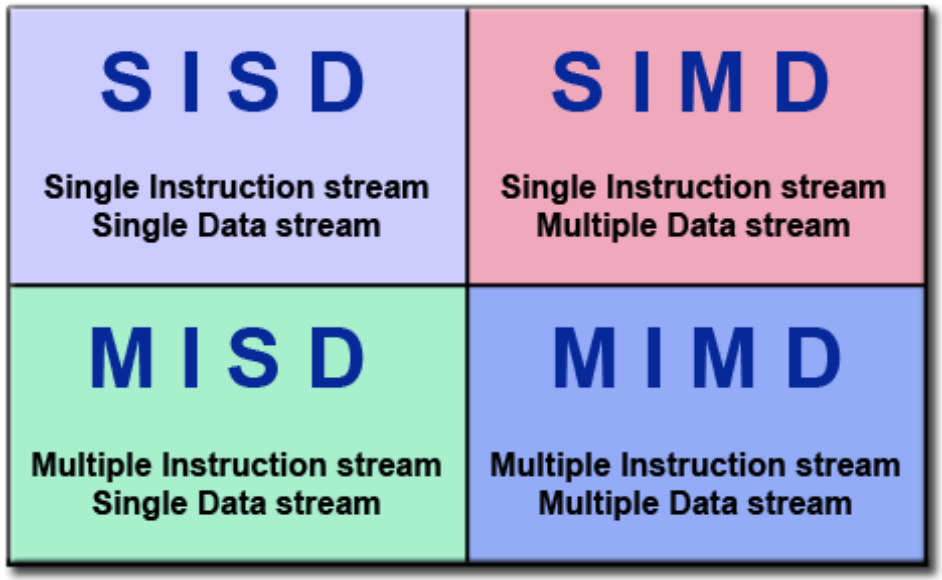


Figure 3: 4 possible classifications according to Flynn

Single Instruction, Single Data (SISD)

A serial (non-parallel) computer

- Single Instruction: Only one instruction stream is being acted on by the CPU during any one clock cycle
- Single Data: Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer

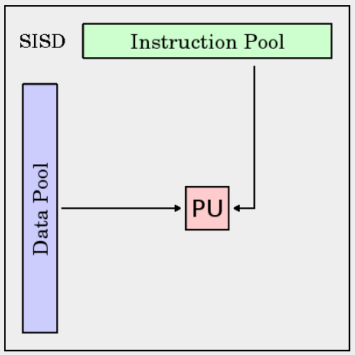
Single Instruction, Single Data (SISD)

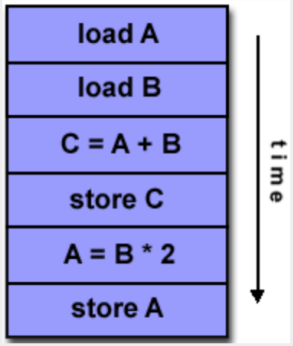
A serial (non-parallel) computer

- Single Instruction: Only one instruction stream is being acted on by the CPU during any one clock cycle
- Single Data: Only one data stream is being used as input during any one clock cycle
- Deterministic execution
- This is the oldest type of computer

- Examples: older generation mainframes, minicomputers, workstations and single processor/core PCs.

Single Instruction, Single Data (SISD)






UNIVAC1


IBM 360


CRAY1


CDC 7600


PDP1


Dell Laptop

Figure 4: Single Instruction, Single Data (SISD)

Single Instruction, Multiple Data (SIMD)

A type of parallel computer

- Single Instruction: All processing units execute the same instruction at any given clock cycle
- Multiple Data: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines

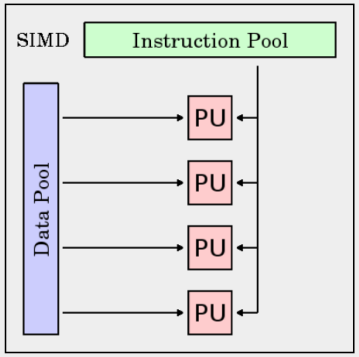
Single Instruction, Multiple Data (SIMD)

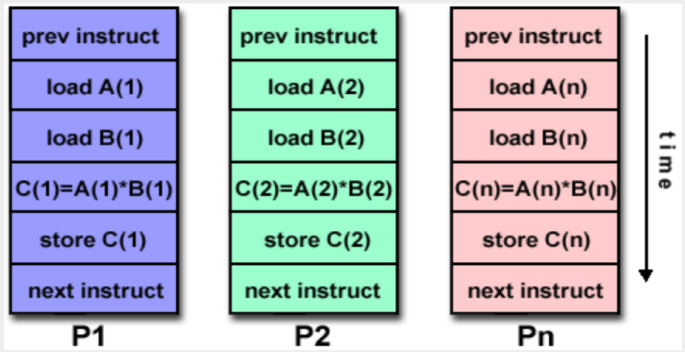
A type of parallel computer

- Single Instruction: All processing units execute the same instruction at any given clock cycle
- Multiple Data: Each processing unit can operate on a different data element
- Best suited for specialized problems characterized by a high degree of regularity, such as graphics/image processing.
- Synchronous (lockstep) and deterministic execution
- Two varieties: Processor Arrays and Vector Pipelines

- Examples:
  - Processor Arrays: Thinking Machines CM-2, MasPar MP-1 & MP-2, ILLIAC IV
  - Vector Pipelines: IBM 9000, Cray X-MP, Y-MP & C90, Fujitsu VP, NEC SX-2, Hitachi S820, ETA10
- Most modern computers, particularly those with graphics processor units (GPUs)

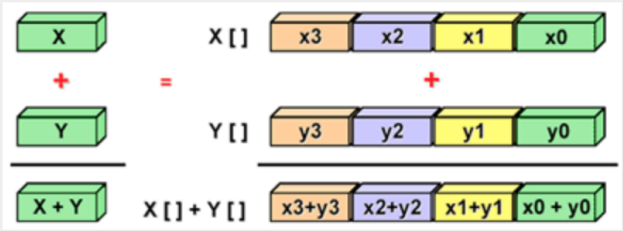
Single Instruction, Multiple Data (SIMD)





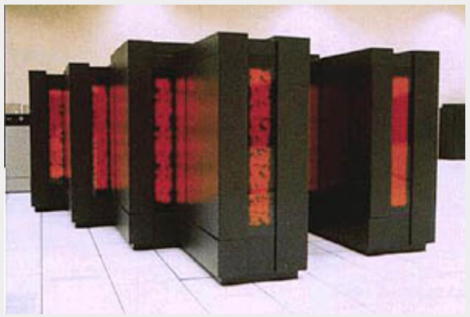
ILLIAC IV

MasPar

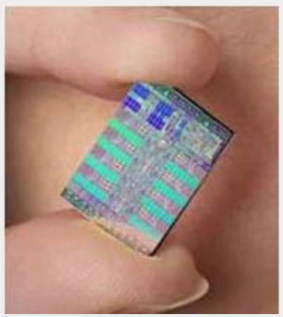


Cray X-MP

Cray Y-MP



Thinking Machines CM-2



Cell Processor (GPU)

Figure 5: Single Instruction, Multiple Data (SIMD)

Multiple Instruction, Single Data (MISD)

A type of parallel computer

- Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
- Single Data: A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.

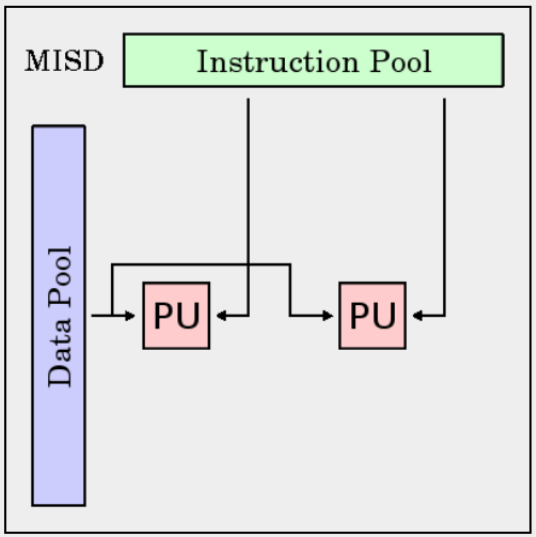
Multiple Instruction, Single Data (MISD)

A type of parallel computer

- Multiple Instruction: Each processing unit operates on the data independently via separate instruction streams.
- Single Data: A single data stream is fed into multiple processing units.
- Few (if any) actual examples of this class of parallel computer have ever existed.

- Some conceivable uses might be:
  - multiple frequency filters operating on a single signal stream
  - multiple cryptography algorithms attempting to crack a single coded message.

Multiple Instruction, Single Data (MISD)



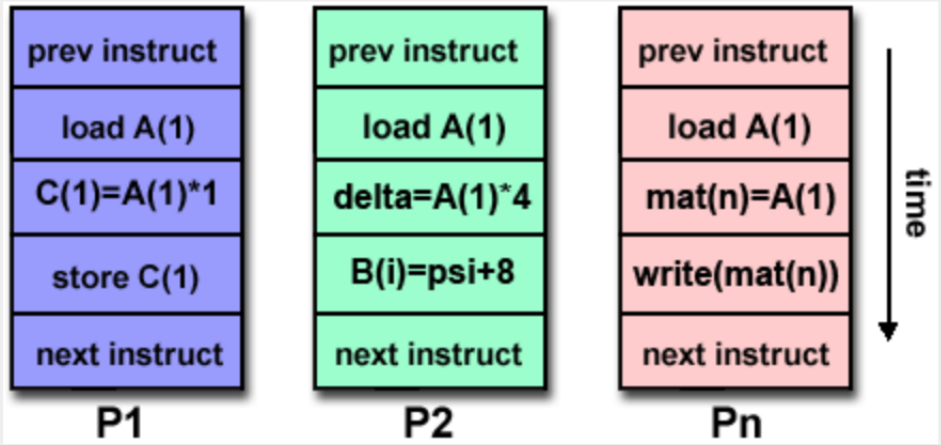


Figure 6: Multiple Instruction, Single Data (MISD)

Multiple Instruction, Multiple Data (SIMD)

A type of parallel computer

- Multiple Instruction: Every processor may be executing a different instruction stream
- Multiple Data: Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.

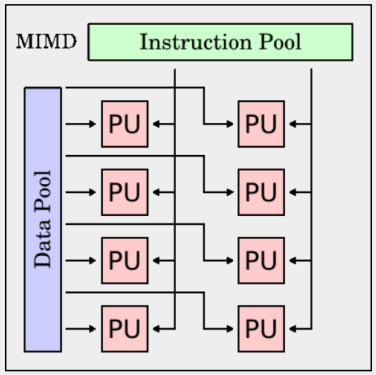
Multiple Instruction, Multiple Data (SIMD)

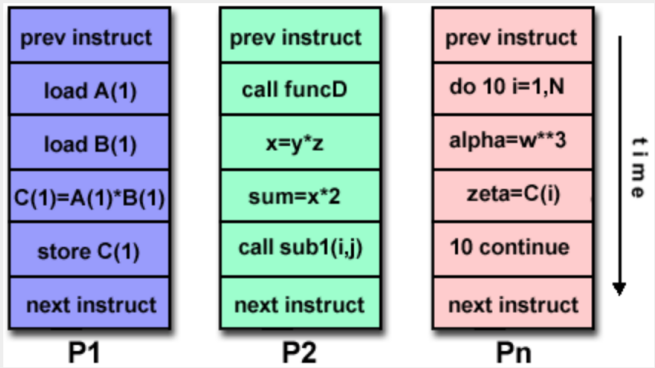
A type of parallel computer

- Multiple Instruction: Every processor may be executing a different instruction stream
- Multiple Data: Every processor may be working with a different data stream
- Execution can be synchronous or asynchronous, deterministic or non-deterministic
- Currently, the most common type of parallel computer - most modern supercomputers fall into this category.

- Examples: most current supercomputers, networked parallel computer clusters and "grids", multi-processor SMP computers, multi-core PCs.
- Note: many MIMD architectures also include SIMD execution sub-components

Multiple Instruction, Multiple Data (SIMD)





IBM POWER5

HP/Compaq Alphaserver

Intel IA32

AMD Opteron

Cray XT3

IBM BG/L

Figure 7: Multiple Instruction, Multiple Data (SIMD)

Some General Parallel Terminology

- Like everything else, parallel computing has its own "jargon". Some of the more commonly used terms associated with parallel computing are listed below.
- Most of these will be discussed in more detail later.

Supercomputing / High Performance Computing (HPC) Using the world's fastest and largest computers to solve large problems.

Node A standalone "computer in a box". Usually comprised of multiple CPUs/processors/cores, memory, network interfaces, etc.
Nodes are networked together to produce a supercomputer.

Some General Parallel Terminology

CPU / Socket / Processor / Core This varies, depending upon who you talk to.

In the past, a CPU (Central Processing Unit) was a singular execution component for a computer.

Then, multiple CPUs were incorporated into a node.

Then, individual CPUs were subdivided into multiple "cores", each being a unique execution unit. CPUs with multiple cores are sometimes called "sockets" - vendor dependent.

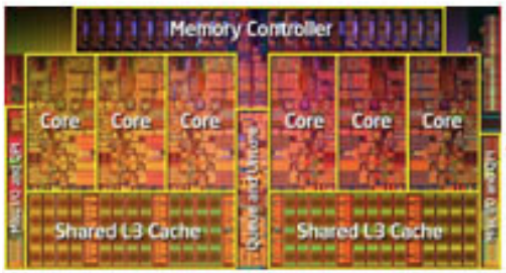
The result is a node with multiple CPUs, each containing multiple cores. The nomenclature is confused at times. Wonder why?

Some General Parallel Terminology

Supercomputer - each blue light is a node

Node - standalone Von Neumann computer

CPU / Processor / Socket - each has multiple cores / processors.



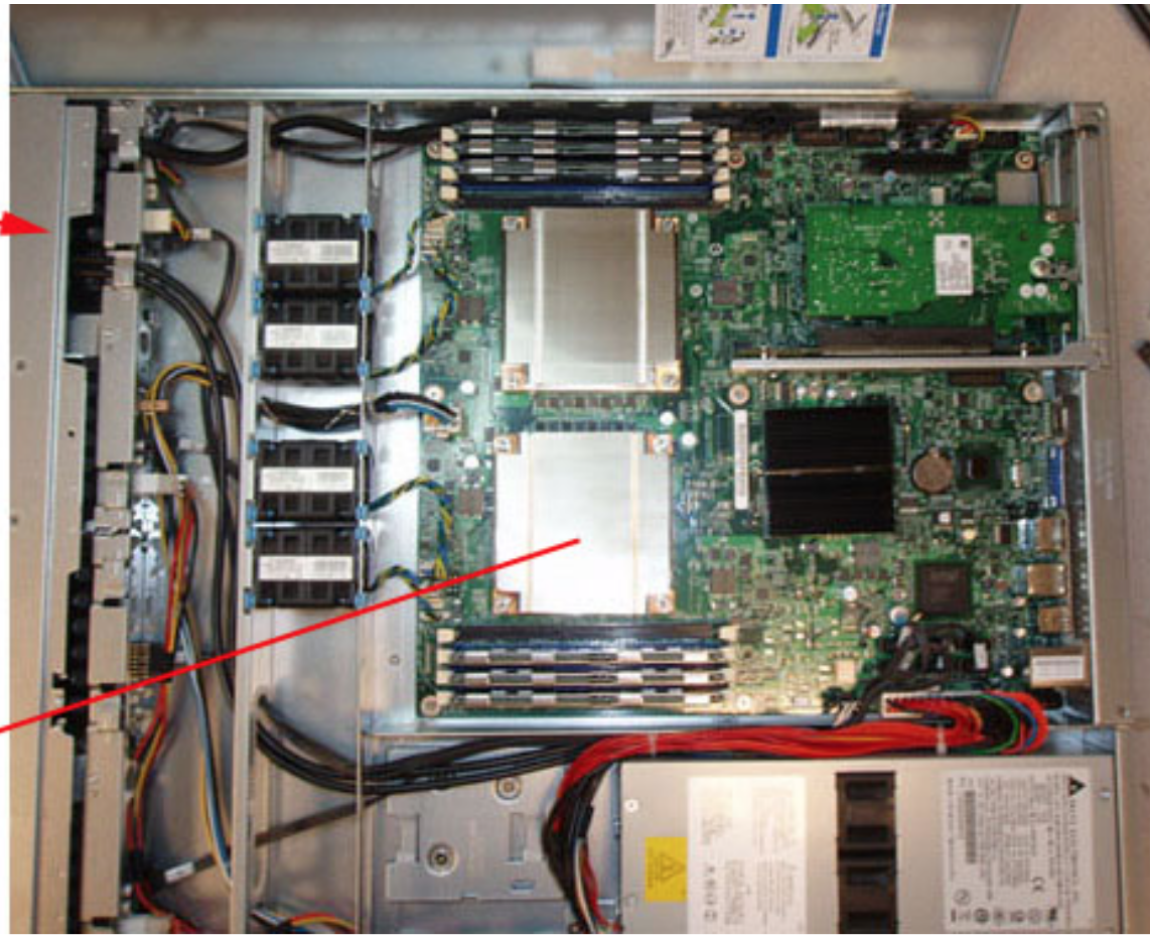


Figure 8: CPU / Socket / Processor / Core

Some General Parallel Terminology

Task A logically discrete section of computational work. A task is typically a program or program-like set of instructions that is executed by a processor. A parallel program consists of multiple tasks running on multiple processors.

Pipelining Breaking a task into steps performed by different processor units, with inputs streaming through, much like an assembly line; a type of parallel computing.

Some General Parallel Terminology

Shared Memory From a strictly hardware point of view, describes a computer architecture where all processors have direct (usually bus based) access to common physical memory. In a programming sense, it describes a model where parallel tasks all have the same "picture" of memory and can directly address and access the same logical memory locations regardless of where the physical memory actually exists.

Symmetric Multi-Processor (SMP) Shared memory hardware architecture where multiple processors share a single address space and have equal access to all resources.

Some General Parallel Terminology

Distributed Memory In hardware, refers to network based memory access for physical memory that is not common.
As a programming model, tasks can only logically "see" local machine memory and must use communications to access memory on other machines where other tasks are executing.

Communications Parallel tasks typically need to exchange data.
There are several ways this can be accomplished, such as through a shared memory bus or over a network, however the actual event of data exchange is commonly referred to as communications regardless of the method employed.

Some General Parallel Terminology

Synchronization The coordination of parallel tasks in real time, very often associated with communications.

Often implemented by establishing a synchronization point within an application where a task may not proceed further until another task(s) reaches the same or logically equivalent point.

Synchronization usually involves waiting by at least one task, and can therefore cause a parallel application's wall clock execution time to increase.

Granularity In parallel computing, granularity is a qualitative measure of the ratio of computation to communication.

- Coarse: relatively large amounts of computational work are done between communication events
- Fine: relatively small amounts of computational work are done between communication events

Some General Parallel Terminology

Observed Speedup Observed speedup of a code which has been parallelized, defined as:

$$\frac{\text{wall-clock time of serial execution}}{\text{wall-clock time of parallel execution}}$$

One of the simplest and most widely used indicators for a parallel program's performance.

Parallel Overhead The amount of time required to coordinate parallel tasks, as opposed to doing useful work. Parallel overhead can include factors such as:

- Task start-up time
- Synchronizations
- Data communications
- Software overhead imposed by parallel languages, libraries, operating system, etc.
- Task termination time

Some General Parallel Terminology

Massively Parallel Refers to the hardware that comprises a given parallel system - having many processing elements. The meaning of "many" keeps increasing, but currently, the largest parallel computers are comprised of processing elements numbering in the hundreds of thousands to millions.

Embarrassingly Parallel Solving many similar, but independent tasks simultaneously; little to no need for coordination between the tasks.

Scalability Refers to a parallel system's (hardware and/or software) ability to demonstrate a proportionate increase in parallel speedup with the addition of more resources. Factors that contribute to scalability include:

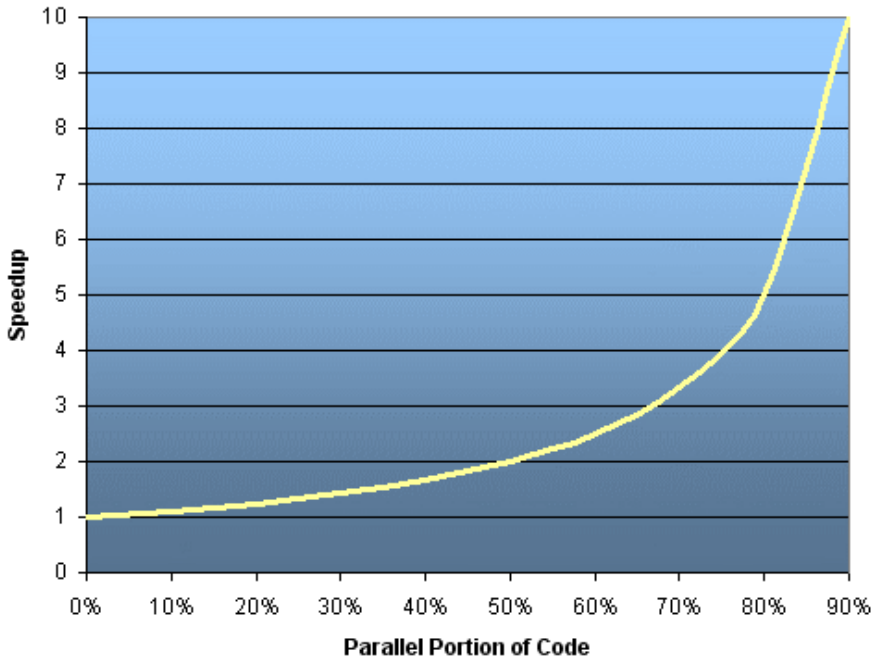
- Hardware - particularly memory-cpu bandwidths and network communication properties
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application

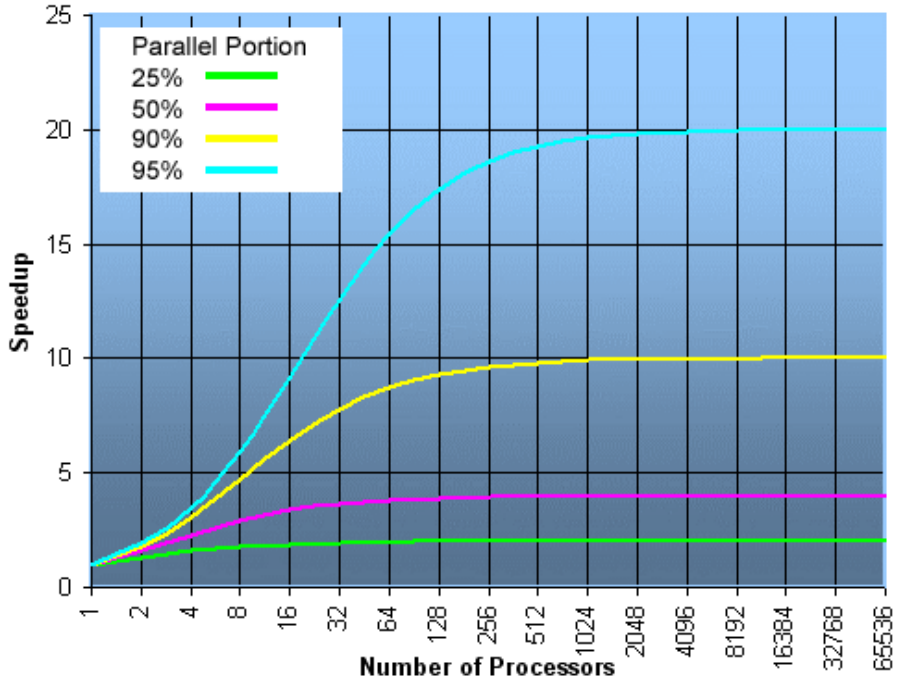
Limits and Costs of Parallel Programming

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 no code can be parallelized, $P = 0$ and speedup $= 1$ (no speedup).
- If all code is parallelized, $P = 1$ and the speedup $= \infty$ (in theory).
- If 50% of code can be parallelized, $max(\text{speedup}) = 2$, meaning the code may run twice as fast.

Amdahl's Law

Introducing the number N 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

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

For example:

	speedup		
N	P = .50	P = .90	P = .99
10	1.82	5.26	9.17
100	1.98	9.17	50.25
1,000	1.99	9.91	90.99
10,000	1.99	9.91	99.02
100,000	1.99	9.99	99.90

Figure 9: Speedup table

Amdahl's Law

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.

Complexity

- In general, 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
  - 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.

Portability

- Thanks to standardization in several APIs, such as MPI, POSIX threads, and OpenMP, portability issues with parallel programs are not as serious as in years past. However...
- 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.

Resource Requirements

- The primary intent of parallel programming is to decrease execution wall clock time, however in order to accomplish this, more CPU time is required. 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 than serial 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.

Scalability

- Two types of scaling based on time to solution:
  - Strong scaling: The total problem size stays fixed as more processors are added.
  - Weak scaling: The problem size per processor stays fixed as more processors are added.
- The ability of a parallel program's performance to scale is a result of a number of interrelated factors. Simply adding more processors is rarely the answer.
- The algorithm may have inherent limits to scalability. At some point, adding more resources causes performance to decrease. Many parallel solutions demonstrate this characteristic at some point.
- Hardware factors play a significant role in scalability. Examples:
  - Memory-cpu bus bandwidth on an SMP machine
  - Communications network bandwidth
  - Amount of memory available on any given machine or set of machines
  - Processor clock speed
- Parallel support libraries and subsystems software can limit scalability independent of your application.

Sequential implementation of domain integration of polynomials

Domain integration of polynomials

Finite formulae for evaluation of integrals:

$$II_S \equiv \int\int_S f(\mathbf{p})\, dS, \qquad III_P \equiv \int\int\int_P f(\mathbf{p})\, dV, \tag{1}$$

The integrating function is a trivariate polynomial

$$f(\mathbf{p}) = \sum_{\alpha=0}^{n} \sum_{\beta=0}^{m} \sum_{\gamma=0}^{p} a_{\alpha\beta\gamma} x^{\alpha} y^{\beta} z^{\gamma},$$

where α, β, γ are non-negative integers. Since the extension to $f(\mathbf{p})$ is straightforward, we focus on integrals of monomials:

$$II_S^{\alpha\beta\gamma} \equiv \int\int_S x^{\alpha} y^{\beta} z^{\gamma}\, dS, \qquad III_P^{\alpha\beta\gamma} \equiv \int\int\int_P x^{\alpha} y^{\beta} z^{\gamma}\, dV. \tag{2}$$

From Cattani, Paoluzzi. "Boundary integration over linear polyhedra", CAD, 1990

Basic integration functions

structure product over a polyhedral surface S, open or closed, is a summation of structure products (3) over the 2-simplices of a triangulation K_2 of S:

$$II_S^{\alpha\beta\gamma} = \int\int_S x^\alpha y^\beta z^\gamma \, dS = \sum_{\tau \in K_2} II_\tau^{\alpha\beta\gamma}$$

```
function II(P, alpha, beta, gamma, signedInt=false)
    w = 0
    V, FV = P
    if typeof(P) == PyCall.PyObject
        if typeof(V) == Array{Any,2}
            V = V'
        end
        if typeof(FV) == Array{Any,2}
            FV = [FV[k,:] for k=1:size(FV,1)]
            FV = FV+1
        end
    end
    if typeof(FV) == Array{Int64,2}
        FV = [FV[:,k] for k=1:size(FV,2)]
    end
    for i=1:length(FV)
        tau = hcat([V[:,v] for v in FV[i]]...)
        if size(tau,2) == 3
            term = TT(tau, alpha, beta, gamma, signedInt)
            if signedInt
                w += term
            else
                w += abs(term)
            end
        elseif size(tau,2) > 3
            println("ERROR: FV[$(i)] is not a triangle")
        else
            println("ERROR: FV[$(i)] is degenerate")
        end
    end
    return w
```

Basic integration functions

$$III_P^{\alpha\beta\gamma} = \int\int\int_P x^\alpha y^\beta z^\gamma \, dx \, dy \, dz$$

$$= \frac{1}{\alpha + 1} \sum_{\tau \in K_2} \left[\frac{(\mathbf{a} \times \mathbf{b})_x}{|\mathbf{a} \times \mathbf{b}|} \right]_\tau II_\tau^{\alpha+1,\beta,\gamma}$$

```
function III(P, alpha, beta, gamma)
    w = 0
    V, FV = P
    if typeof(P) == PyCall.PyObject
        if typeof(V) == Array{Any,2}
            V = V'
        end
        if typeof(FV) == Array{Any,2}
            FV = [FV[k,:] for k=1:size(FV,1)]
            FV = FV+1
        end
    end
    for i=1:length(FV)
        tau = hcat([V[:,v] for v in FV[i]]...)
        vo,va,vb = tau[:,1],tau[:,2],tau[:,3]
        a = va - vo
        b = vb - vo
        c = cross(a,b)
        w += c[1]/vecnorm(c) * TT(tau, alpha+1, beta, gamma)
    end
    return w/(alpha + 1)
end
```

Basic integration functions

$$
\begin{aligned}
II_{\tau}^{\alpha\beta\gamma} \quad = \quad & |\mathbf{a}\times\mathbf{b}| \sum_{h=0}^{\alpha} \binom{\alpha}{h} x_o^{\alpha-h} \cdot \\
& \cdot \sum_{k=0}^{\beta} \binom{\beta}{k} y_o^{\beta-k} \cdot \\
& \cdot \sum_{m=0}^{\gamma} \binom{\gamma}{m} z_o^{\gamma-m} \cdot \\
& \cdot \sum_{i=0}^{h} \binom{h}{i} a_x^{h-i} b_x^{i} \cdot \\
& \cdot \sum_{j=0}^{k} \binom{k}{j} a_y^{k-j} b_y^{j} \cdot \\
& \cdot \sum_{l=0}^{m} \binom{m}{l} a_z^{m-l} b_z^{l} \cdot \\
& \cdot II^{\mu\nu}
\end{aligned}
$$

```
function TT(tau::Array{Float64,2}, alpha, beta, gamma, signedInt=false)
  vo,va,vb = tau[:,1],tau[:,2],tau[:,3]
  a = va - vo
  b = vb - vo
  s1 = 0.0
  for h=0:alpha
    for k=0:beta
      for m=0:gamma
        s2 = 0.0
        for i=0:h
          s3 = 0.0
          for j=0:k
            s4 = 0.0
            for l=0:m
              s4 += binomial(m,l) * a[3]^(m-l) * b[3]^l *
                M( h+k+m-i-j-l, i+j+l )
            end
            s3 += binomial(k,j) * a[2]^(k-j) * b[2]^j * s4
          end
          s2 += binomial(h,i) * a[1]^(h-i) * b[1]^i * s3;
        end
        s1 += binomial(alpha,h) * binomial(beta,k) * binomial(gamma,m) *
            vo[1]^(alpha-h) * vo[2]^(beta-k) * vo[3]^(gamma-m) * s2
      end
    end
  end
  c = cross(a,b)
  if signedInt == true
    return s1 * vecnorm(c) * sign(c[3])
  else return s1 * vecnorm(c) end
end
```

Basic integration functions

$$I^{\alpha\beta} = \frac{1}{\alpha+1} \sum_{h=0}^{\alpha+1} \binom{\alpha+1}{h} \frac{(-1)^h}{h+\beta+1},$$

```
function M(alpha, beta)
    a = 0
    for l=1:(alpha + 2)
        a += binomial(alpha+1,l) * (-1)^l/(l+beta+1)
    end
    return a/(alpha + 1)
end
```