

## Parallel & Distributed Computing: Lecture 6

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

October 11, 2018

# Concepts and Terminology

- 1 Some General Parallel Terminology
- 2 Limits and Costs of Parallel Programming

## Some General Parallel Terminology

# Synchronization & Granularity

**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.

# Synchronization & Granularity

**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.

# Synchronization & Granularity

**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

# Synchronization & Granularity

**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

# Observed Speedup & Parallel Overhead

**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.



# Observed Speedup & Parallel Overhead

**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:

# Observed Speedup & Parallel Overhead

**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

# Observed Speedup & Parallel Overhead

**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

# Observed Speedup & Parallel Overhead

**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

# Observed Speedup & Parallel Overhead

**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.

# Observed Speedup & Parallel Overhead

**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

# Massively & Embarrassingly Parallel; Scalability

**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**.

# Massively & Embarrassingly Parallel; Scalability

**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.



# Massively & Embarrassingly Parallel; Scalability

**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:

# Massively & Embarrassingly Parallel; Scalability

**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

# Massively & Embarrassingly Parallel; Scalability

**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

# Massively & Embarrassingly Parallel; Scalability

**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

# Massively & Embarrassingly Parallel; Scalability

**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:

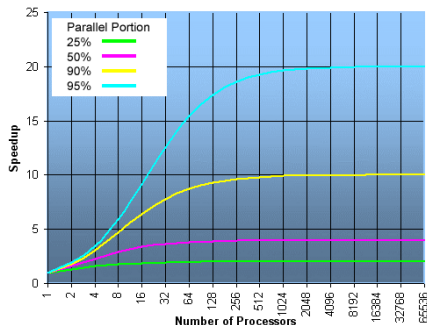
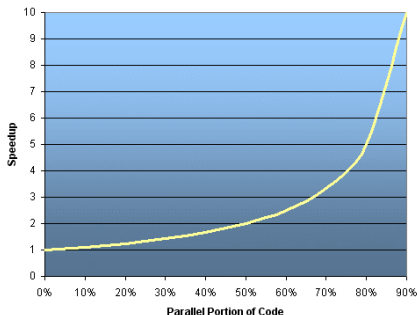
- Hardware - particularly memory-cpu bandwidths and network communication properties
- Application algorithm
- Parallel overhead related
- Characteristics of your specific application

# Limits and Costs of Parallel Programming

# Amdahl's Law 1/4

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

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

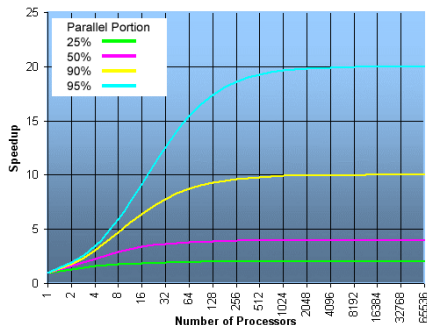
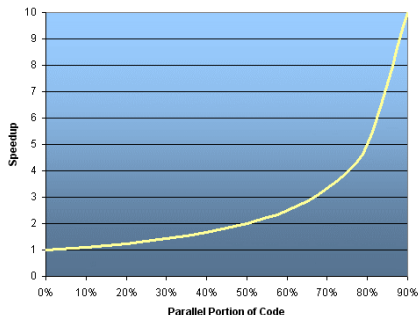


- If no code can be parallelized,  $P = 0$  and speedup = 1 (no speedup).

# Amdahl's Law 1/4

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

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



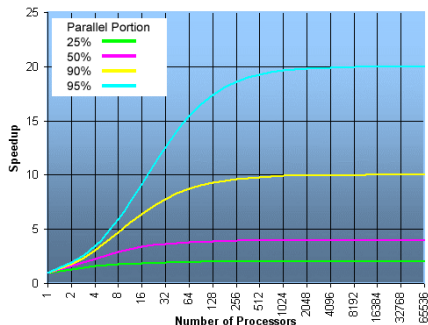
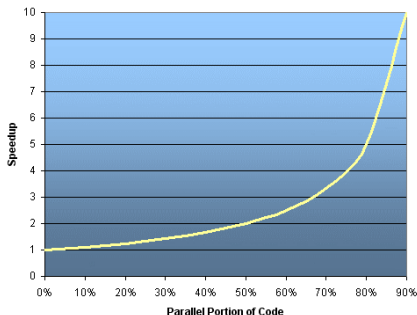
- If no code can be parallelized,  $P = 0$  and speedup = 1 (no speedup).
- If all code is parallelized,  $P = 1$  and the speedup =  $\infty$  (in theory).



# Amdahl's Law 1/4

**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 2/4

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 3/4

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

For example:

N	speedup		
	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 1: **Speedup table**

# Amdahl's Law 4/4

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.

# 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:

# 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

# 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



# 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

# 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

# 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

# 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. . .

# 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.

# 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.

# 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.



# 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.

# 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.

# 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 1/2

- Two types of scaling based on time to solution:

# Scalability 1/2

- Two types of scaling based on time to solution:
  - **Strong scaling:** The total problem size stays fixed as more processors are added.

# Scalability 1/2

- 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.

# Scalability 1/2

- 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**.



## Scalability 2/2

- 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.

## Scalability 2/2

- 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:

## Scalability 2/2

- 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

## Scalability 2/2

- 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**

## Scalability 2/2

- 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

## Scalability 2/2

- 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**

## Scalability 2/2

- 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.