CMPT 450/750: Computer Architecture
Fall 2022

Parallel Architectures

Alaa Alameldeen & Arrvindh Shriraman

© Copyright 2022 Alaa Alameldeen and Arrvindh Shriraman

Sequential vs. Parallel Execution

Sequential

(single stream of instructions)

OP [operand, …]
OP [operand, …]
OP [operand, …]
OP [operand, …]
OP [operand, …]
OP [operand, …]
OP [operand, …]

Parallel

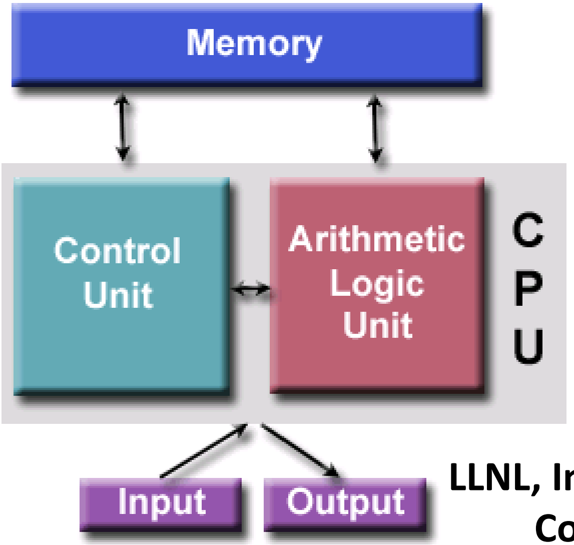
(multiple independent instruction streams)

OP [operand, …]	OP [operand, …]
OP [operand, …]	OP [operand, …]
OP [operand, …]	OP [operand, …]
OP [operand, …]	OP [operand, …]
OP [operand, …]	OP [operand, …]
OP [operand, …]	OP [operand, …]
OP [operand, …]	OP [operand, …]

- **Single-core systems:** Goal is to achieve best single-thread performance
- **Multi-core (or multi-processor) systems:** Goal is to achieve best parallel performance for all threads

von Neumann Architecture

- **Uses the stored-program concept**
  - The CPU executes a stored program that specifies a sequence of memory read and write operations

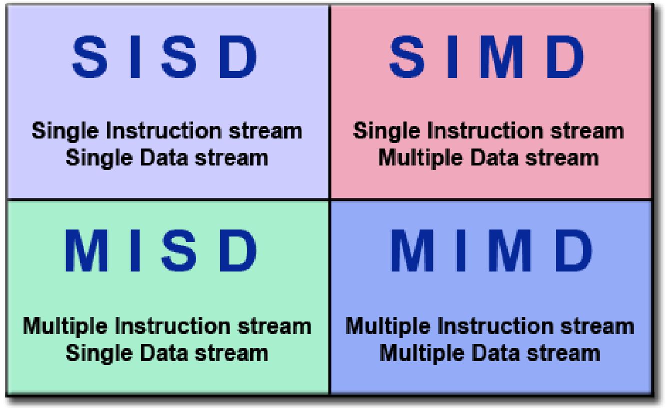


LLNL, Introduction to Parallel Computing Tutorial

- **Basic design:**
  - Memory stores both program and data
  - Program instructions are coded data which instruct the computer on what to do
  - Data: Information to be used by the program
  - CPU gets instructions & data from memory, decodes instructions and then performs them sequentially

Multiprocessor Taxonomy (Flynn)

- **Instructions and Data streams can be either single or multiple**
- **Single Instruction, Single Data (SISD)**
  - ➢Serial, non-parallel computer – e.g., single CPU PCs, workstations and mainframes
- **Single Instruction, Multiple Data (SIMD)**
  - ➢All processor units execute same instruction on different data
  - ➢Example: Vector processors such as IBM 9000, Cray C90
- **Multiple Instruction, Single Data (MISD)**
  - ➢Rare (e.g., C.mmp), data operated on by multiple instructions
- **Multiple Instruction, Multiple Data (MIMD)**
  - ➢Most modern parallel computers
  - ➢Processors may be executing different instructions on different data streams

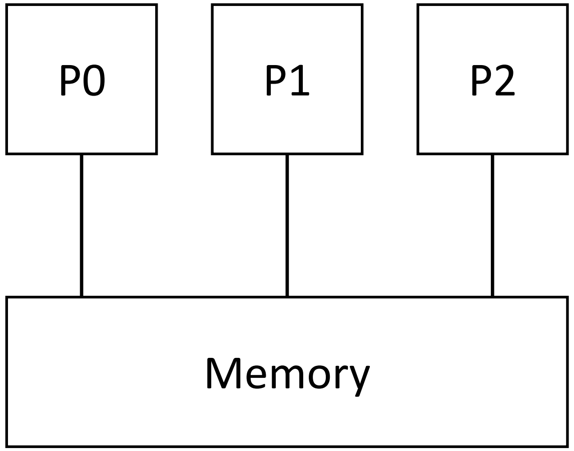


SISD Single Instruction stream Single Data stream	SIMD Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	MIMD Multiple Instruction stream Multiple Data stream

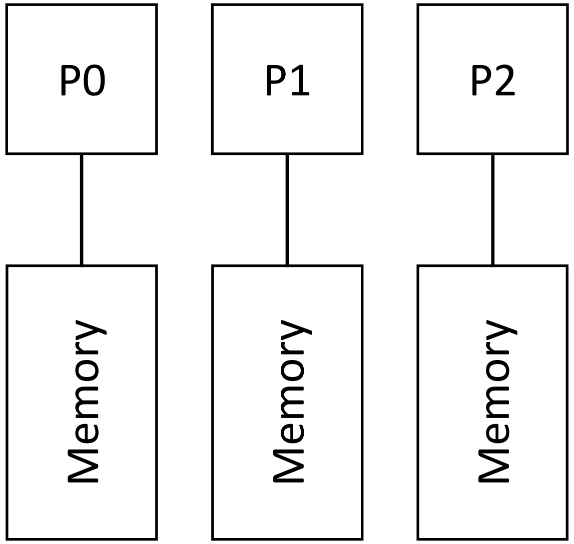
LLNL, Introduction to Parallel Computing Tutorial

Memory Architectures

Shared Memory

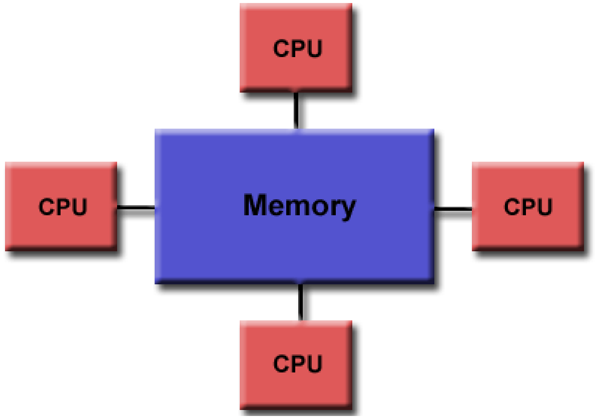


Distributed Memory

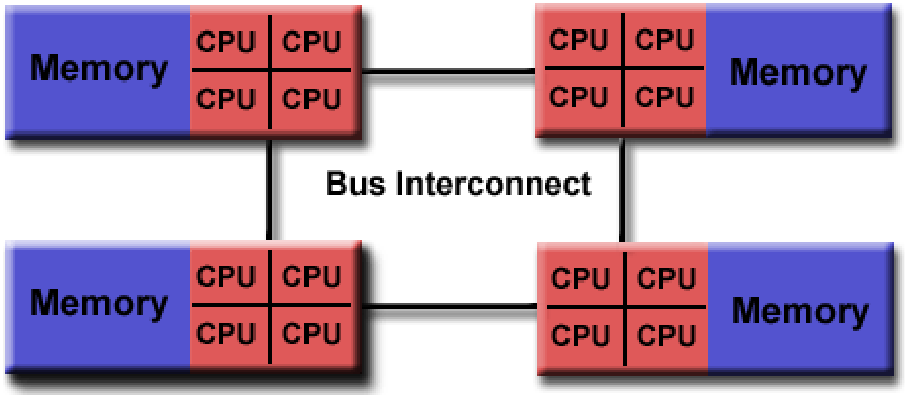


Shared Memory Architecture

- All processors can access all memory
- Processors share memory resources, but can operate independently
- One processor's memory changes are seen by all other processors
- Uniform Memory Access (UMA) Architecture
  - ➢ Example: Symmetric Multiprocessor (SMP) machines
  - ➢ Identical processors with equal access and equal access time to memory
  - ➢ Also called CC-UMA - Cache Coherent UMA. Cache coherent means that if one processor updates a location in shared memory, all the other processors know about the update

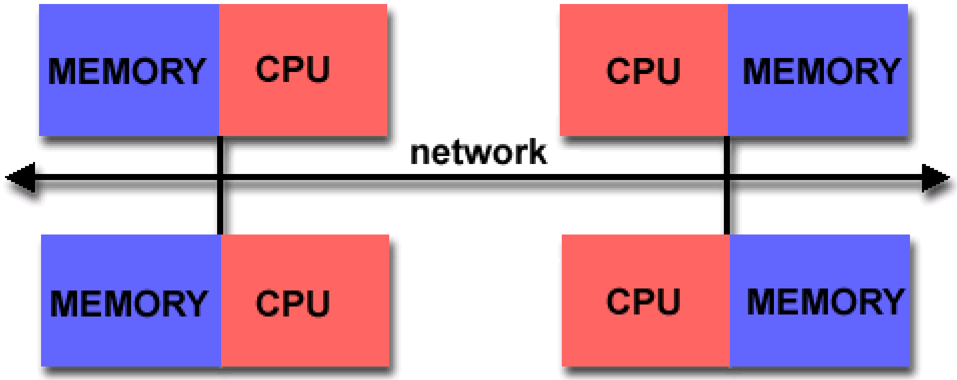


- Non-Uniform Memory Access (NUMA) Architecture
  - ➢ Often made by physically linking two or more SMPs
  - ➢ One processor can directly access memory of another processor
  - ➢ Not all processors have equal access time to all memories
  - ➢ Memory access across link is slower
  - ➢ Called CC-NUMA - Cache Coherent NUMA - if CC is maintained
- Pros and Cons?



LLNL, Introduction to Parallel Computing Tutorial

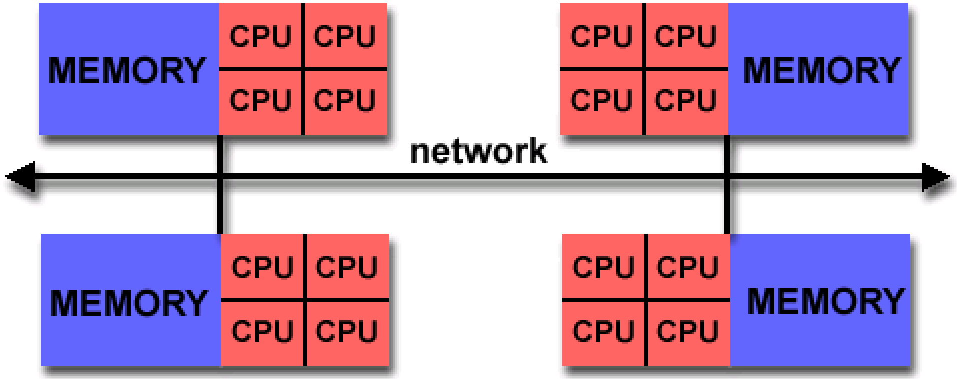
Distributed Memory Architecture



LLNL, Introduction to Parallel Computing Tutorial

- **Require a communication network to connect inter-processor memory**
- **Processors have their own local memory**
  - ➢ Memory addresses in one processor do not map to another processor
  - ➢ No concept of global address space across all processors.
- **Each processor operates independently, changes to its local memory have no effect on the memory of other processors**
  - ➢ Cache coherence does not apply
- **Programs explicitly define how and when data is communicated and how tasks are synchronized**
- **Pros and cons?**

Hybrid Distributed-Shared Memory Architecture



LLNL, Introduction to Parallel Computing Tutorial

- **Shared memory component is usually a CC SMP**
  - ➢ Processors on a given SMP can address that machine's memory as global memory
- **Distributed memory component is the networking of multiple SMPs**
  - ➢ SMPs know only about their own memory
  - ➢ Network communications needed to move data between SMPs
- **Largest, fastest computers share this architecture**

Sequential vs. Parallel Execution

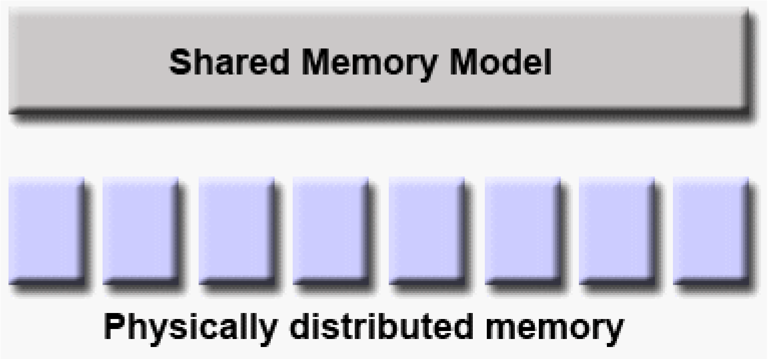
```
1 withdraw(int account_id, int amount) {
2  balance = get_balance(account_id);
3  if(balance >= amount) {
4         set_balance(account_id, balance - amount);
5         eject(amount);
6    }
7 }
```

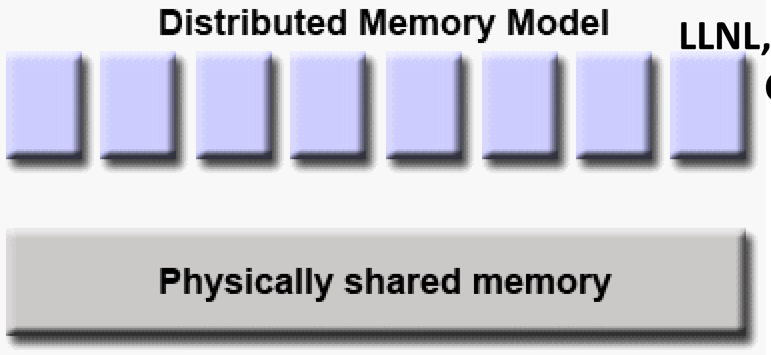
withdraw(5, 900);	withdraw(5, 700);
2 balance = 1000	
	2 balance = 1000
3 balance >= 900	3 balance >= 700
4 balance = 100	
	4 balance = -600

Time ↓

Parallel Programming Models

- **Parallel programming models:**
  - ➢ Shared Memory
  - ➢ Threads
  - ➢ Message Passing
  - ➢ Data Parallel
  - ➢ Hybrid
- **Parallel programming models exist as an abstraction above hardware and memory architectures**
- **Models not specific to certain types of memory architecture or machine**
  - ➢ Shared memory can be implemented on a distributed memory architecture
  - ➢ Message passing can be implemented on a shared memory architecture (e.g., SGI Origin)

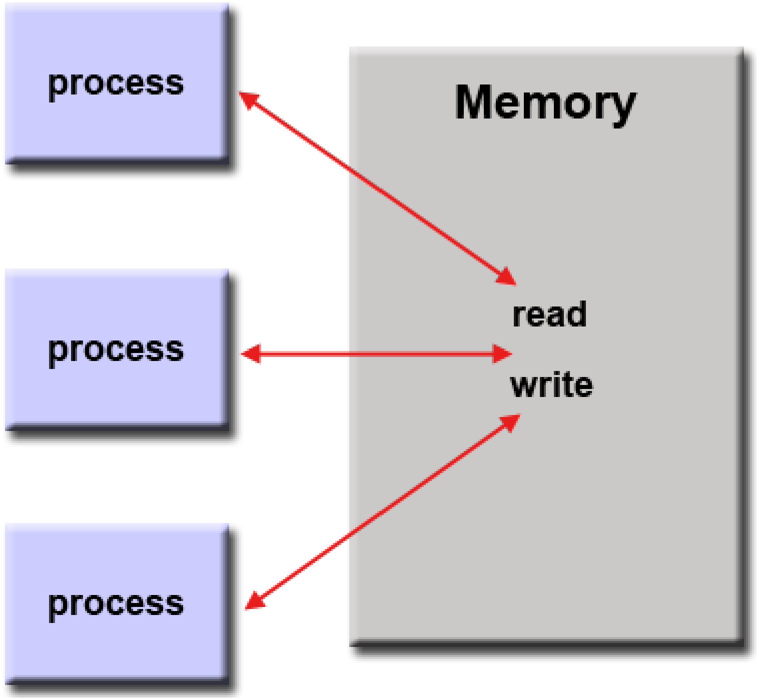




LLNL, Introduction to Parallel Computing Tutorial

Shared Memory Model

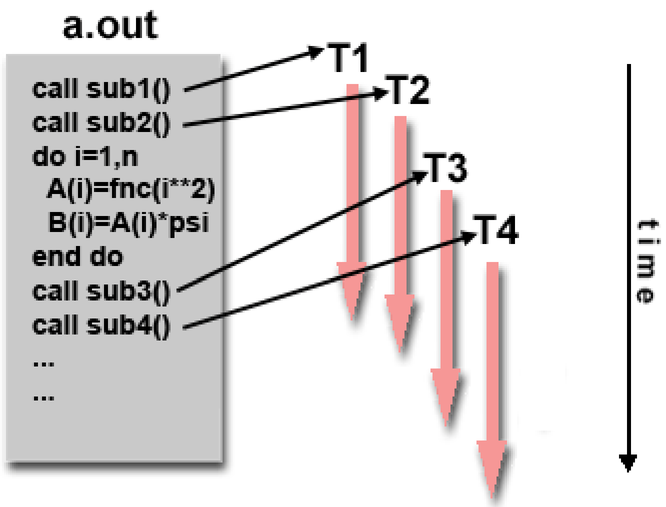
- Tasks share a common address space, which they read and write asynchronously
- Various mechanisms such as locks / semaphores may be used to control access to the shared memory
- Relatively simple programming model
  - No notion of data "ownership“
  - No need to specify explicit communication of data between tasks
- But it’s more difficult to understand and manage data locality
  - We don’t know OR we can’t take advantage of memory being physically closer to a particular process



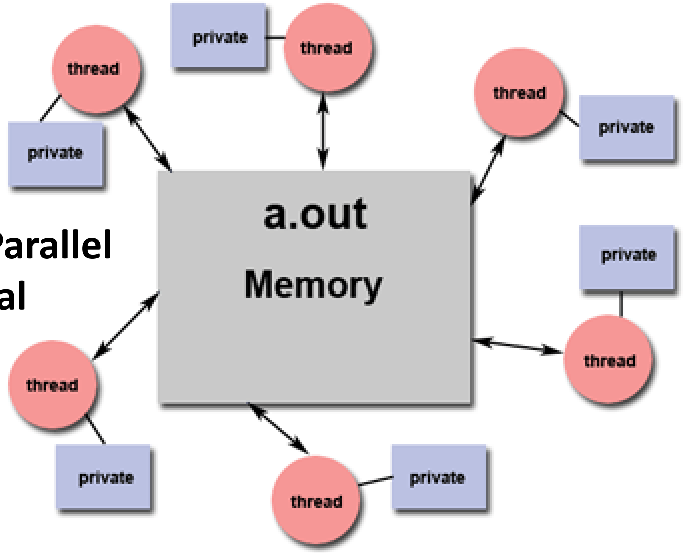
LLNL, Introduction to Parallel Computing Tutorial

Threads Model

- Single process can have multiple, concurrent execution paths
- Analogy: a single program that includes a number of subroutines
  - Program creates a number of tasks (threads) that can be scheduled and run by the operating system concurrently
- Each thread has local data, but also shares all resources of the program (e.g., memory space)
- Threads communicate with each other through global memory by updating address locations
  - This requires synchronization constructs to ensure that more than one thread is not updating the same global address at the same time
- Threads are commonly associated with shared memory architectures and operating systems
- Implementation standards
  - POSIX threads (Pthreads): Library-based, available in C
    - Tutorial: https://computing.llnl.gov/tutorials/pthreads/
  - OpenMP: Compiler-directive-based
    - Tutorial: https://computing.llnl.gov/tutorials/openMP/

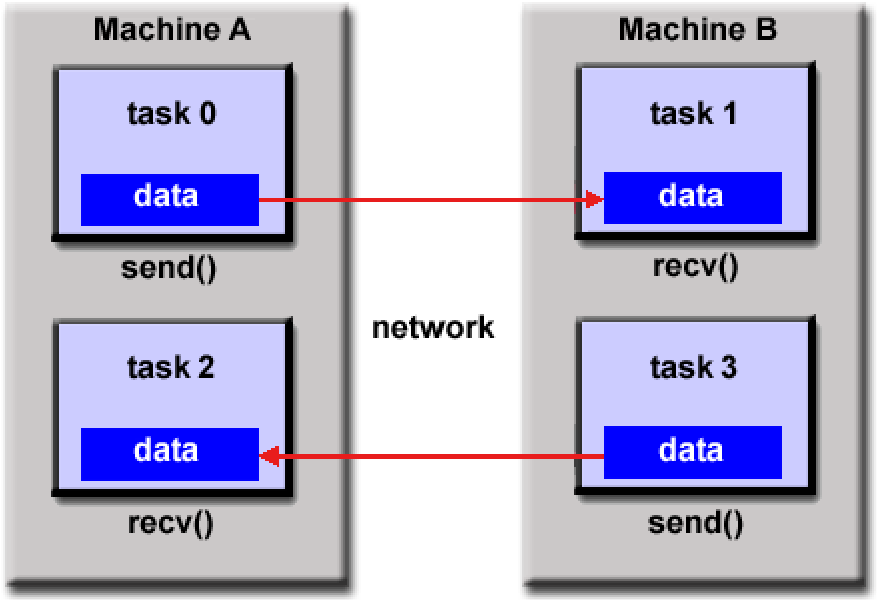


```
call sub1()
call sub2()
do i=1,n
  A(i)=fnc(i**2)
  B(i)=A(i)*psi
end do
call sub3()
call sub4()
...
...
```



LLNL, Introduction to Parallel Computing Tutorial

Message Passing Model

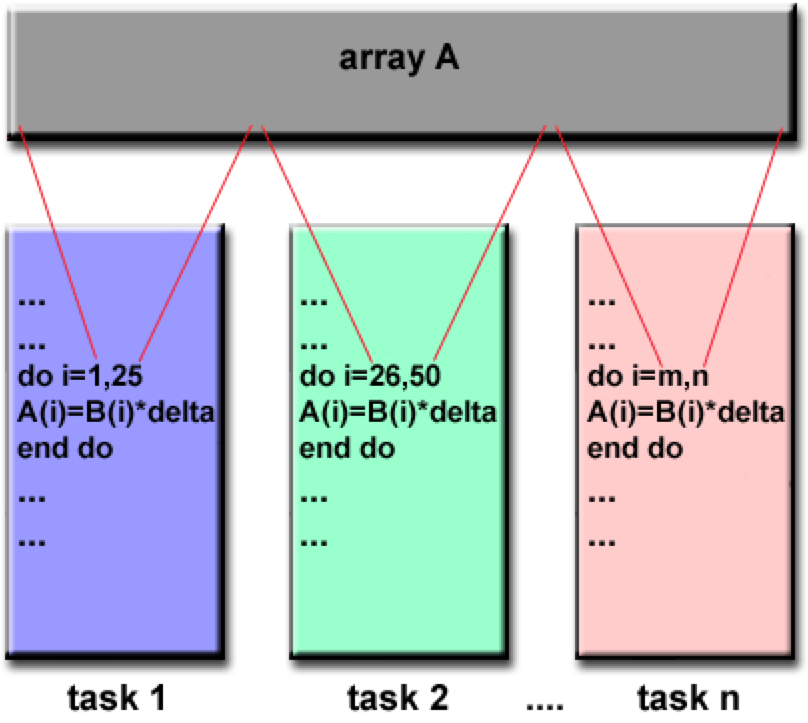


LLNL, Introduction to Parallel Computing Tutorial

- **Tasks use their own local memory during computation**
- **Multiple tasks can run on the same physical machine OR across an arbitrary number of machines**
- **Tasks exchange data through communications by sending and receiving messages**
  - ➢ Requires cooperative operations by each process, e.g., a send operation must have a matching receive operation
- **Implementations**
  - ➢ Library of subroutines that are embedded in source code. The programmer is responsible for determining all parallelism.
  - ➢ Message Passing Interface (MPI) is the current industry standard for message passing
    - ❑ Tutorial: https://computing.llnl.gov/tutorials/mpi/
  - ➢ For shared memory architectures, MPI implementations use shared memory (memory copies) for task communications

Data Parallel Model

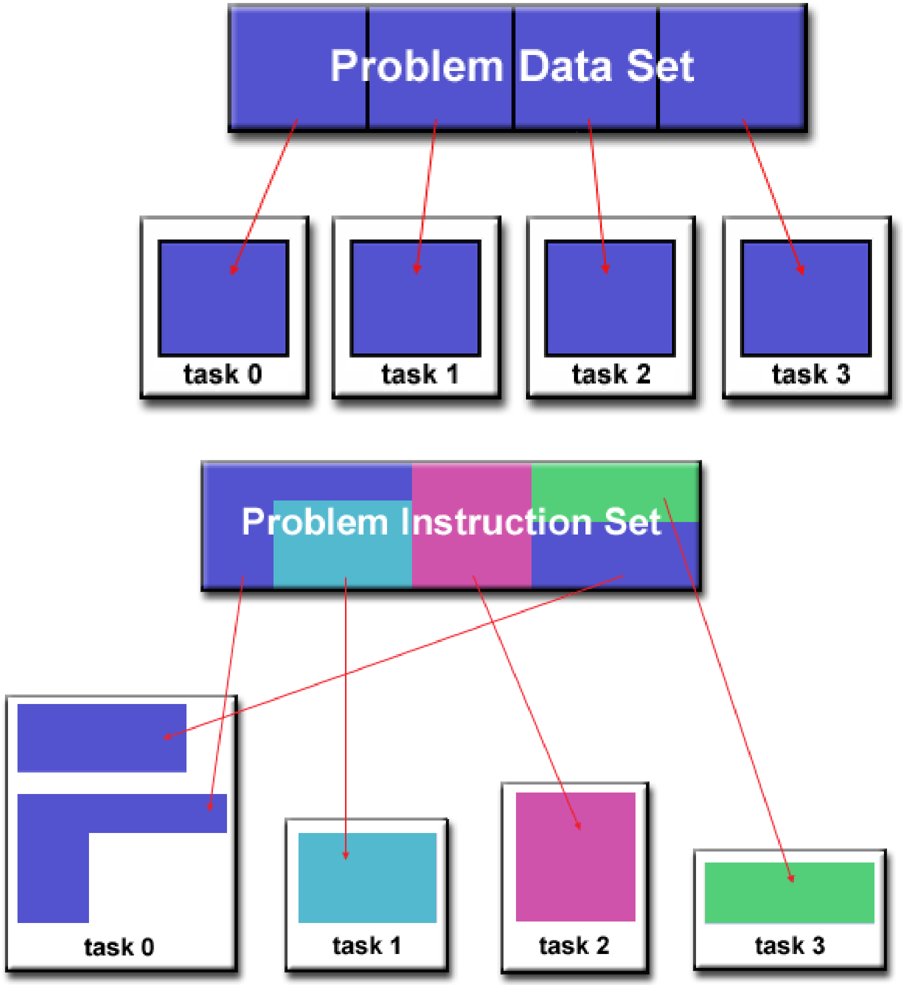
- **Most parallel work focuses on performing operations on a data set**
  - ➢ Data set is typically organized into a common structure, such as an array or cube
- **Each task works on a different partition of the same data structure, all tasks work collectively on the structure**
- **Tasks perform the same operation on their partition of work, e.g., "add 4 to every array element"**
- **On shared memory architectures, all tasks may have access to the data structure through global memory**
- **On distributed memory architectures the data structure is split up and resides as "chunks" in the local memory of each task**



LLNL, Introduction to Parallel Computing Tutorial

Designing Parallel Programs: Partitioning

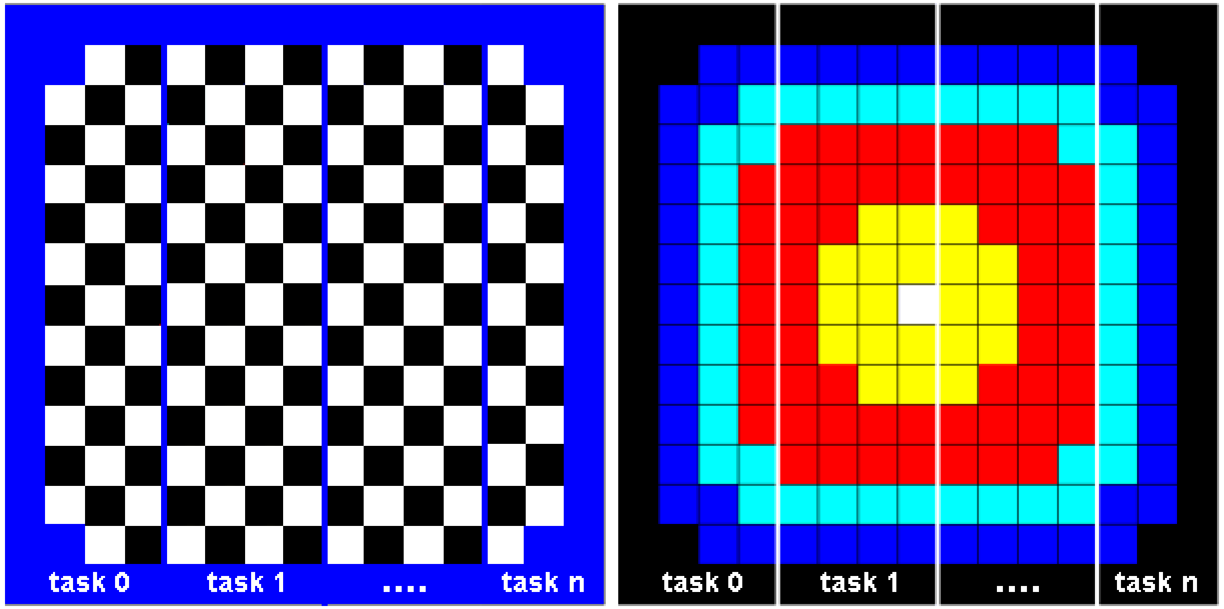
- **Breaking the problem into discrete "chunks" of work that can be distributed to multiple tasks**
  - ➢ Two main types: domain decomposition and functional decomposition
- **Domain Decomposition**
  - ➢ Data is decomposed, each task works on portion of data
  - ➢ Different ways to partition data
- **Functional Decomposition**
  - ➢ The problem is decomposed according to the work that must be done, each task performs a portion of the overall work
- **Programmer can combine both types**



LLNL, Introduction to Parallel Computing Tutorial

Designing Parallel Programs: Communication

- **Some problems can be decomposed and executed in parallel with virtually no data sharing, called *embarrassingly parallel* problems**
  - ➢ Example: image processing operation where every pixel in a black and white image needs to have its color reversed. The image data can easily be distributed to multiple tasks that then act independently of each other to do their portion of the work
- **Most parallel applications are not quite so simple, and do require data sharing between tasks**
- **Issues to consider when writing a program that needs communications**
  - ➢ Cost of communications
  - ➢ Latency vs. bandwidth
  - ➢ Visibility of communications
  - ➢ Synchronous vs. asynchronous communications
  - ➢ Scope of communications
  - ➢ Efficiency of communications
  - ➢ Overhead and complexity



LLNL, Introduction to Parallel Computing Tutorial

Designing Parallel Programs: Synchronization

- **Barrier (all tasks)**
  - ➢ Each task performs its work until it reaches the barrier then stops (blocks)
  - ➢ When the last task reaches the barrier, all tasks are synchronized
  - ➢ Tasks can then continue or serial work is done
- **Lock / semaphore (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 own the lock / semaphore / flag.
  - ➢ The first task to acquire the lock "sets" it and then it 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 (tasks performing communication operation)**
  - ➢ Coordination is required with the other task(s) participating in the communication, e.g., before a task can perform a “send” operation, it must receive an acknowledgment from the receiving task that it is OK to send.

Designing Parallel Programs: Data Dependencies

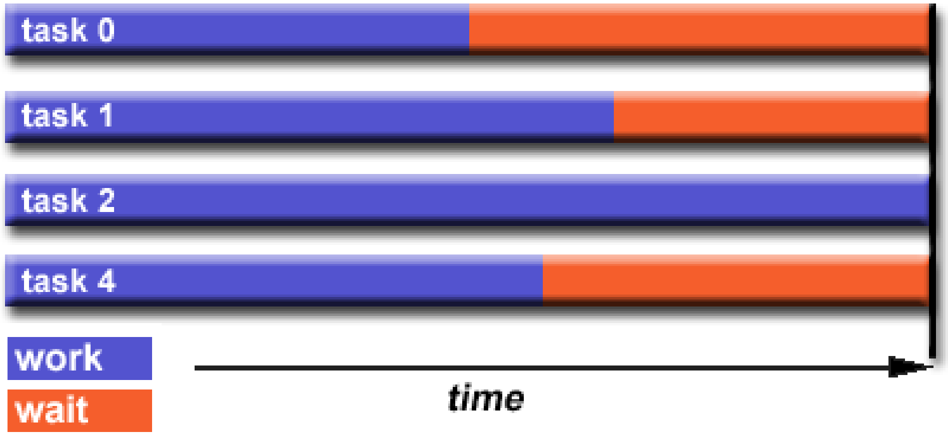
- **A *dependence* exists between program statements when the order of statement execution affects the results of the program**
- **A *data dependence* results from multiple use of the same location(s) in storage by different tasks**
- **Dependencies are one of the primary inhibitors to parallelism**
- **Loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts**

```
for (i = 1; i < N; i++)
        A[i] = A[i-1]*2;
```

- **How to handle data dependencies**
  - Distributed memory architectures: communicate required data at synchronization points
  - Shared memory architectures: synchronize read/write operations between tasks

Designing Parallel Programs: Load Balancing

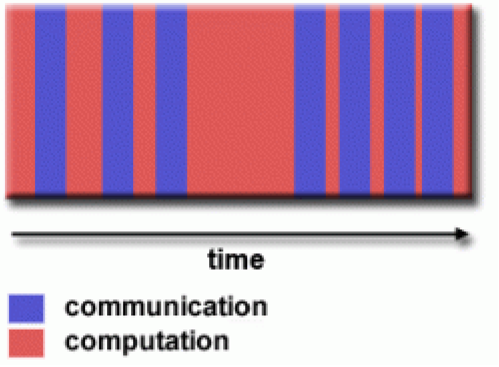
- **Load balancing refers to distributing work among tasks so that all tasks are kept busy all the time**
  - ➢ Minimize idle time
- **Important for performance**
  - ➢ If tasks are synchronized by a barrier, slowest task determines the overall performance
- **How to achieve load balancing**
  - ➢ Equally partition the work each task receives
  - ➢ Dynamic work assignment

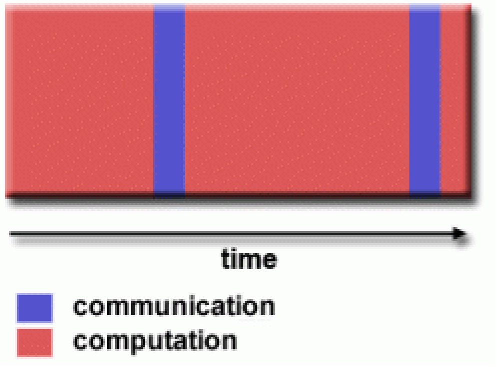


LLNL, Introduction to Parallel Computing Tutorial

Designing Parallel Programs: Granularity

- **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
  - ➢ High communication overhead, less opportunity to improve performance
  - ➢ Overhead required for communications and synchronization between tasks can be longer than the computation
- **Coarse-grain Parallelism**
  - ➢ Relatively large amounts of computational work are done between communication/synchronization events
  - ➢ High computation to communication ratio
  - ➢ Implies more opportunity for performance improvement
  - ➢ Harder to load balance efficiently

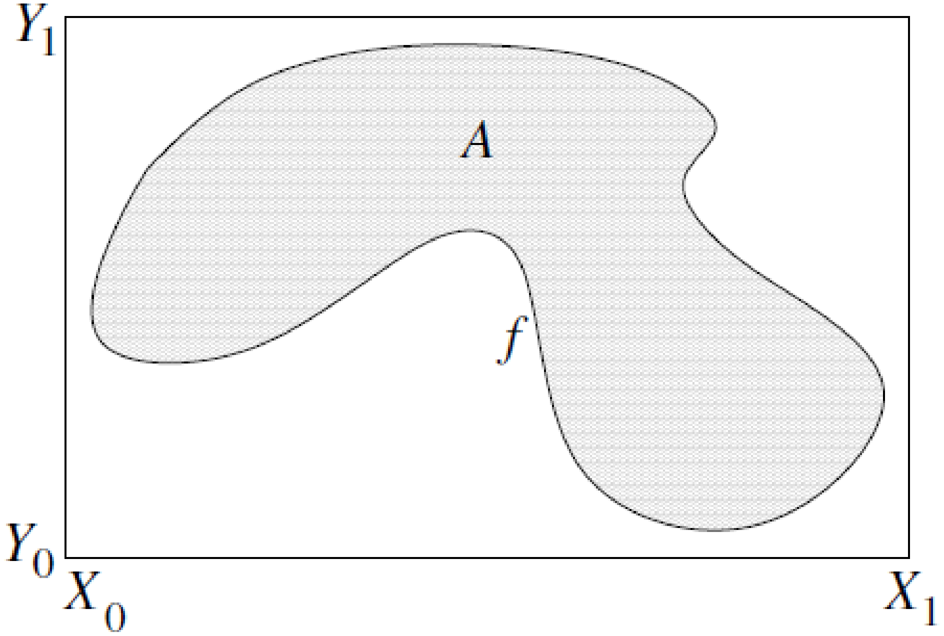




LLNL, Introduction to Parallel Computing Tutorial

Parallel Programs: Examples

Monte Carlo Simulation



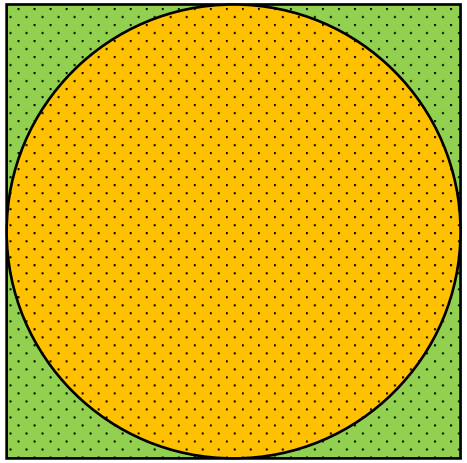
- Allows using randomness as a feature to solve problems that might be deterministic in nature (but don’t have closed form solutions)
- Key insight: With a large enough sample size, the sample mean is a good approximation of the true mean
- Determine Area inside closed curve *f*
- We need a function to determine whether a point is inside *f*
- Simulation Steps:
  - Generate n pairs of random numbers (x_i, y_i) i=1...n such that x_i’s are uniformly distributed in [X_0,X_1] and y_i’s are uniformly distributed in [Y_0, Y_1]
  - Determine total number of points m inside f
  - Approximate area

    $$A = \frac{m}{n} \times (Y_1 - Y_0)(X_1 - X_0)$$
  - Approximation is more accurate when n is large

Monte Carlo π Estimation

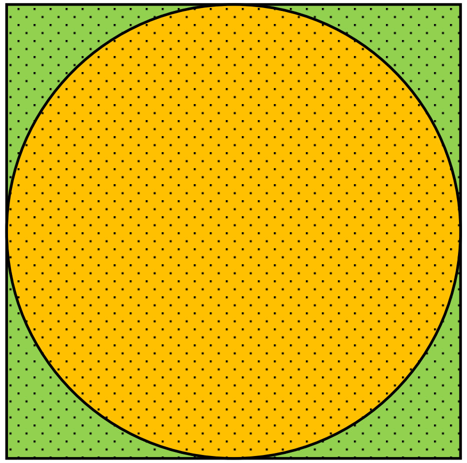
$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

```
circle_count = 0;
for (uint i = 0; i < n; i++) {
        x = get_random(0, 1);
        y = get_random(0, 1);
        if ((x, y) inside circle)
                ++circle_count;
}
pi_value = 4.0 * circle_count / n;
```



Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$



```
circle_count = 0;
create T threads
for each thread in parallel {
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle)
                        ++circle_count;
        }
}
pi_value = 4.0 * circle_count / n;
```

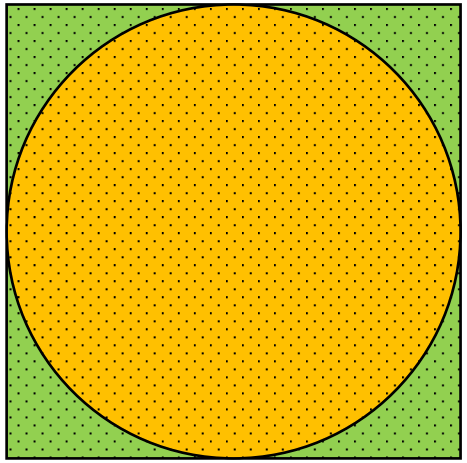
Need to use thread-safe random number generator function

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

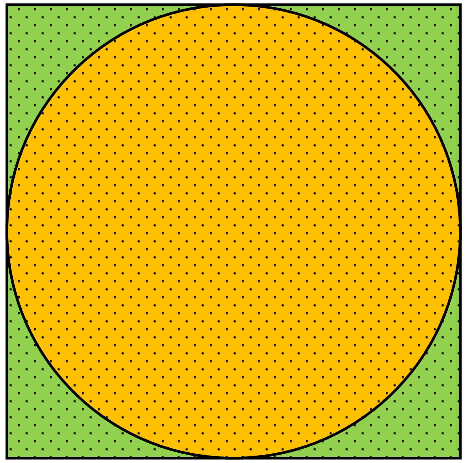
```
circle_count = 0;
create T threads
for each thread in parallel {
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle)
                        ++circle_count;
        }
}
pi_value = 4.0 * circle_count / n;
```

shared variable



Monte Carlo π Estimation

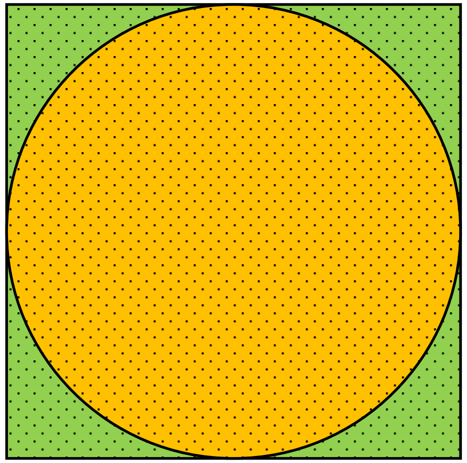
$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$



```
circle_count = 0;
create T threads
for each thread in parallel {
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle) {
                        lock();
                        ++circle_count;
                        unlock();
                }
        }
}
pi_value = 4.0 * circle_count / n;
```

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

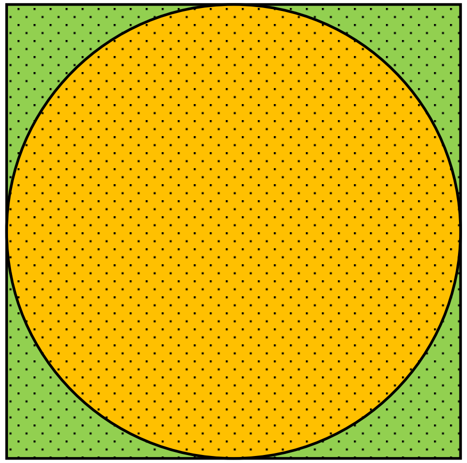


```
circle_count = 0;
create T threads
for each thread in parallel {
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle) {
                        lock();
                        ++circle_count;
                        unlock();
                }
        }
}
pi_value = 4.0 * circle_count / n;
```

serial code

Monte Carlo π Estimation

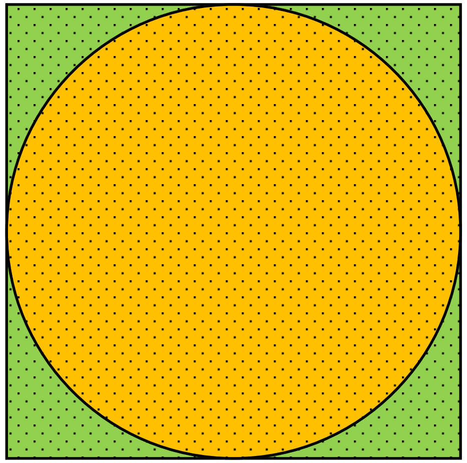
$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$



```
circle_count = 0;
create T threads
for each thread in parallel {
        local_circle_count = 0;
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle)
                        ++local_circle_count;
        }
        lock();
        circle_count += local_circle_count;
        unlock();
}
pi_value = 4.0 * circle_count / n;
```

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

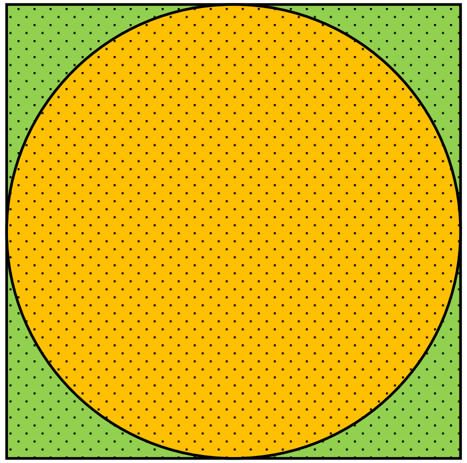


```
circle_count = 0;
create T threads
for each thread in parallel {
        local_circle_count = 0;
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle)
                        ++local_circle_count;
        }
        lock();
        circle_count += local_circle_count;
        unlock();
}
pi_value = 4.0 * circle_count / n;
```

use atomics to eliminate locks

Monte Carlo π Estimation

$$\frac{A(circle)}{A(square)} = \frac{\pi r^2}{4r^2}$$

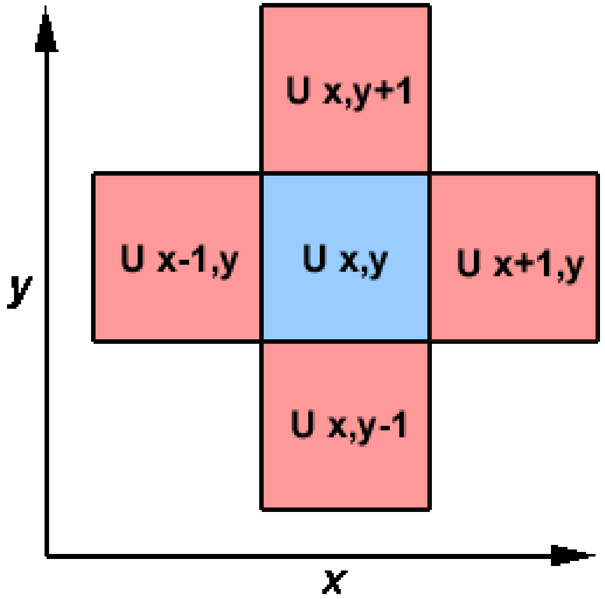


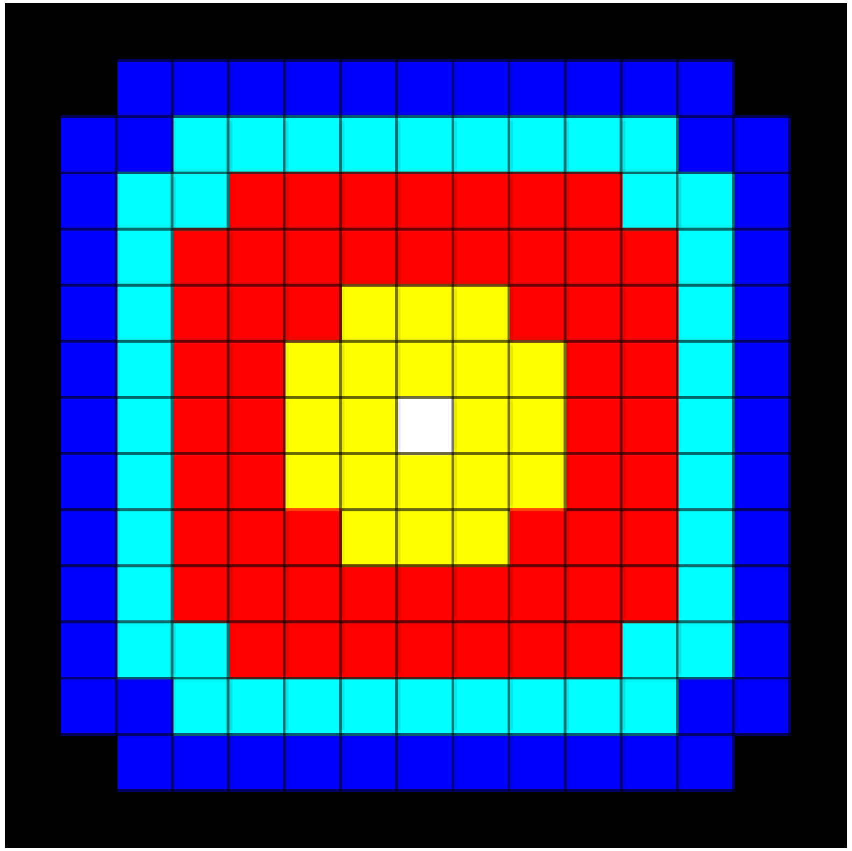
```
circle_count = 0;
create T threads
for each thread in parallel {
        local_circle_count = 0;
        for (uint i = 0; i < approx(n/T); i++) {
                x = get_random(0, 1);
                y = get_random(0, 1);
                if ((x, y) inside circle)
                        ++local_circle_count;
        }
        atomic_add(circle_count, local_circle_count);
}
pi_value = 4.0 * circle_count / n;
```

check out Fetch and Add, Compare and Swap

CAS: https://en.wikipedia.org/wiki/Compare-and-swap
FAA: https://en.wikipedia.org/wiki/Fetch-and-add
C++11 std::atomic: https://en.cppreference.com/w/cpp/atomic

Heat Transfer Problem





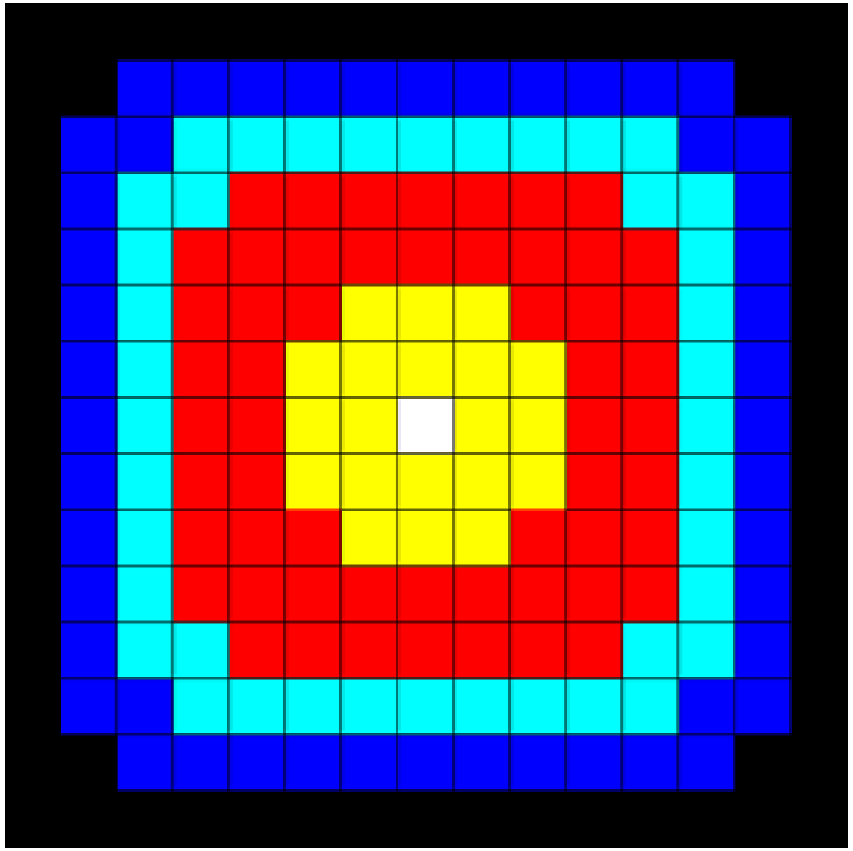
$$U_{x,y} = U_{x,y}$$

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

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

LLNL Parallel Computing Tutorial

Heat Transfer Problem

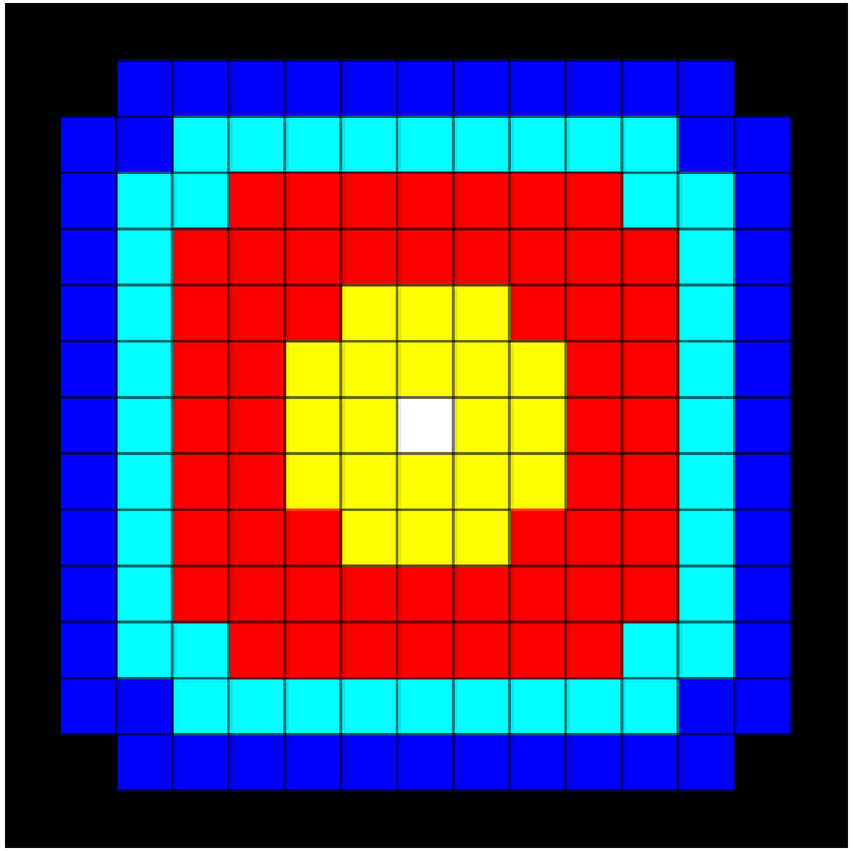


Serial Program:

```
float U[maxX][maxY];
// initialize U
for (int t = 1; t<= MAX_TIME; t++) {
            for (int x=0; x < maxX; x++) {
                        for (int y = 0; y < maxY; y++) {
                                    U[x][y] = U[x][y]
                                    + Cx * (U[x+1][y] + U[x-1][y] – 2*U[x][y])
                                    + Cy * (U[x][y+1] + U[x][y-1] – 2*U[x][y]);
                        }   // for y
            }  // for x
}  // for t
```

Some U values read from current iteration; others read from previous iterations: Incorrect

Heat Transfer Problem



Serial Program:

```
float U1[maxX][maxY];  // values for previous time step
float U2[maxX][maxY];  // values for current time step
for (int t = 1; t<= MAX_TIME; t++) {
            for (int x=0; x < maxX; x++) {
                for (int y = 0; y < maxY; y++) {
                       U2[x][y] = U1[x][y]
                       + Cx * (U1[x+1][y] + U1[x-1][y] – 2*U1[x][y])
                       + Cy * (U1[x][y+1] + U1[x][y-1] – 2*U1[x][y]);
                }   // for y
            }  // for x
            // Swap U1, U2;
}  // for t
```

parallelize

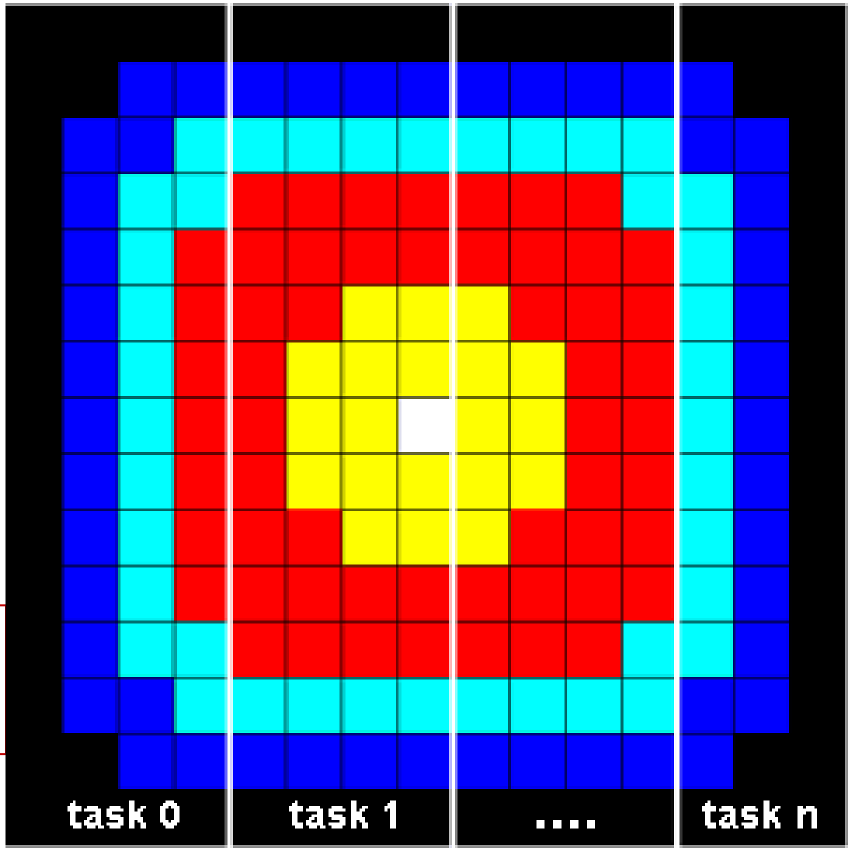
Heat Transfer Problem

Parallel Program:

```
float U1[maxX][maxY];  // values for previous time step
float U2[maxX][maxY];  // values for current time step
create T threads
for each thread in parallel
   for (int t = 1; t<= MAX_TIME; t++) {
            for (x in subset(tid)) {
                 for (int y = 0; y < maxY; y++) {
                          U2[x][y] = U1[x][y]
                          + Cx * (U1[x+1][y] + U1[x-1][y] – 2*U1[x][y])
                          + Cy * (U1[x][y+1] + U1[x][y-1] – 2*U1[x][y]);
                 }   // for y
            }  // for x
            // Swap U1, U2;
   }  // for t
}
```

Still Safe? Think Boundary elements

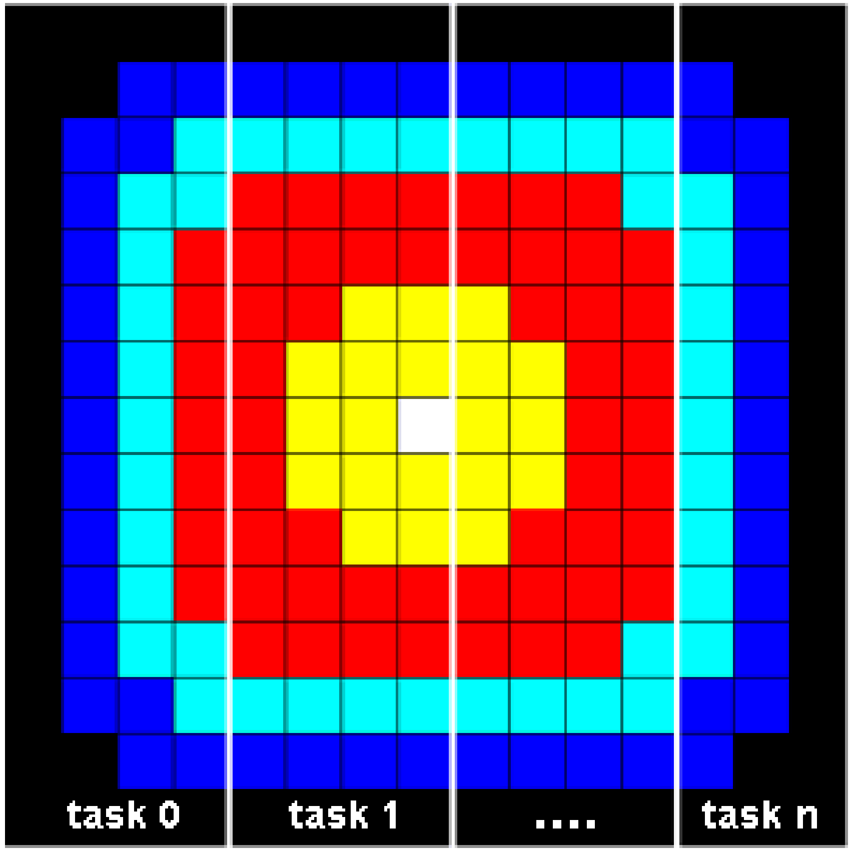
Who does this?



Heat Transfer Problem

Parallel Program:

```
float U1[maxX][maxY];  // values for previous time step
float U2[maxX][maxY];  // values for current time step
create T threads
for each thread in parallel
   for (int t = 1; t<= MAX_TIME; t++) {
            for (x in subset(tid)) {
                 for (int y = 0; y < maxY; y++) {
                           U2[x][y] = U1[x][y]
                           + Cx * (U1[x+1][y] + U1[x-1][y] – 2*U1[x][y])
                           + Cy * (U1[x][y+1] + U1[x][y-1] – 2*U1[x][y]);
                 }   // for y
            }  // for x
            barrier();
            if (tid==0) // Swap U1, U2;
            else // wait till U2 and U1 are swapped
    }  // for t
}
```



Barrier: https://en.wikipedia.org/wiki/Barrier_(computer_science)

Parallel Architectures: Performance Metrics and Limits

Measuring Performance

- **Latency: How fast does a job compute? Elapsed time**
  - ➢Time for computation + communication + synchronization
- **Throughput: How many jobs complete in a given time?**
- **Scalability: How well does the system scale?**
  - ➢Strong Scaling: How the solution time varies with the number of processors (or compute nodes) for a *fixed total problem size*?
  - ➢Weak Scaling: How the solution time varies with the number of processors (or compute nodes) for a *fixed problem size per processor*?

Weak v/s strong scaling: https://en.wikipedia.org/wiki/Scalability#Weak_versus_strong_scaling

Performance Metrics

- **Speedup,** $S_p = \frac{\text{Execution time on 1} - \text{processor system } (T_1)}{\text{Execution time on p} - \text{processor system } (Tp)}$
- **Alternatively,** $S_p = \frac{\text{Throughput on p processors } (X_p)}{\text{Throughput on 1 processor } (X_1)}$
- **Efficiency** **=** $\frac{S_p}{p}$ **(higher is better)**

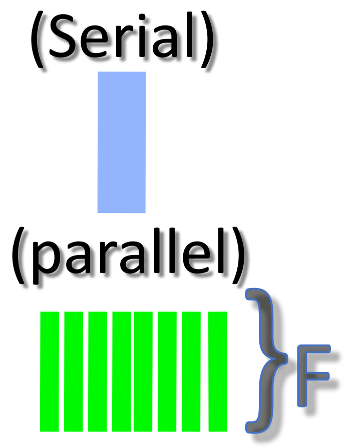
Amdahl's Law [1967, Gene Amdahl]

- **F = fraction of problem that is parallel**
  - ➢(1 – F) = fraction of problem that is sequential
- **Time on 1 processor =** $\frac{1-F}{1}+\frac{F}{1}=1$
- **Time on *p* processors =** $\frac{1-F}{1}+\frac{F}{p}$
- **Speedup with p processors:**

$$S_p = \frac{1}{1-F+\frac{F}{p}}$$

- **Fastest parallel time,** $T_p = T_1 \, x \, (1 - F + \frac{F}{P})$

Program Phases



(Serial)

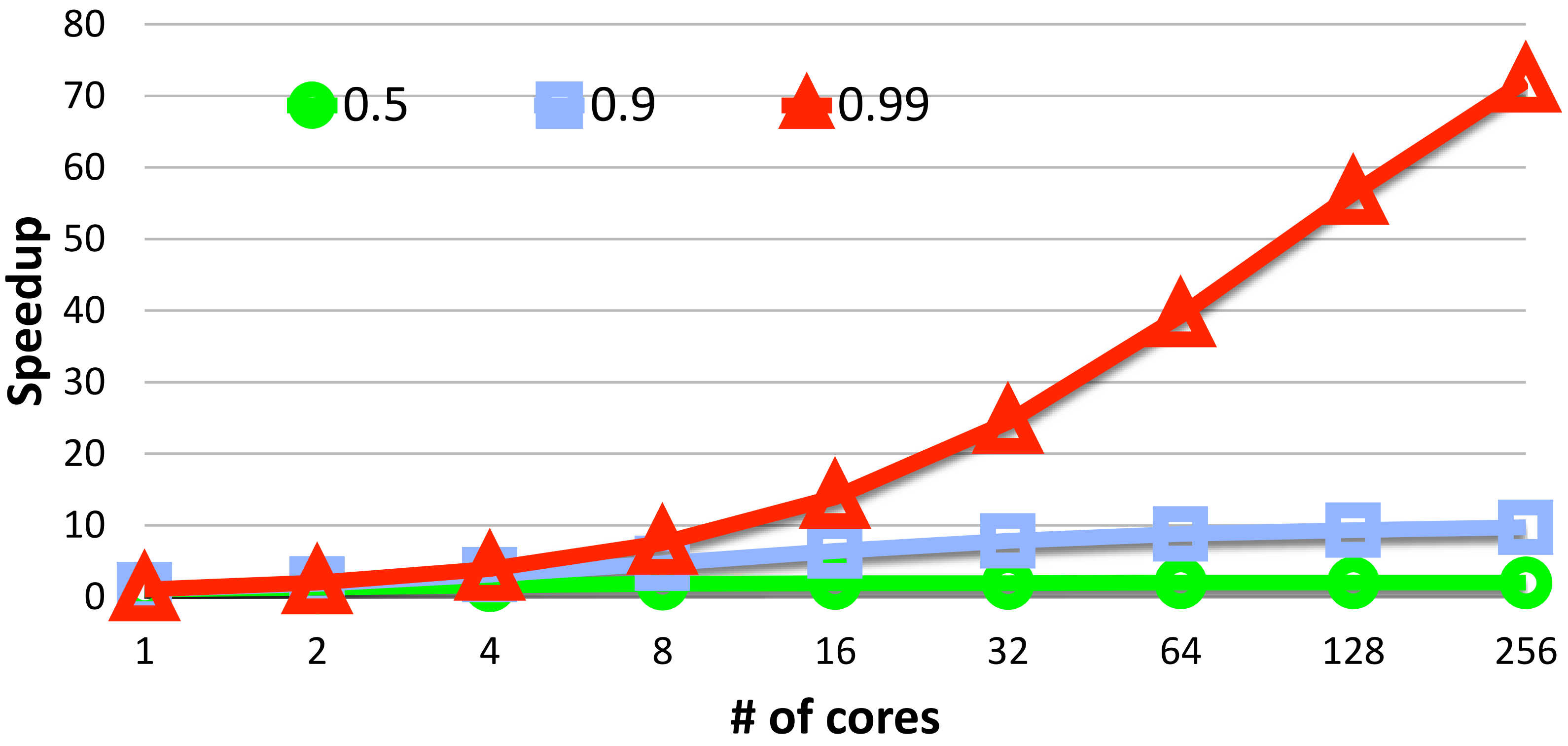
(parallel)

F

Implications of Amdahl's Law

1. **As the parallel part increases, Speedup also increases: S_p ↑ as F ↑**
   - Implies that programmers should focus on parallelizing as much of the program as possible

Parallel Speedup



Implications of Amdahl's Law

1. As the parallel part increases, Speedup also increases: S_p ↑ as F ↑

- Implies that programmers should focus on parallelizing as much of the program as possible

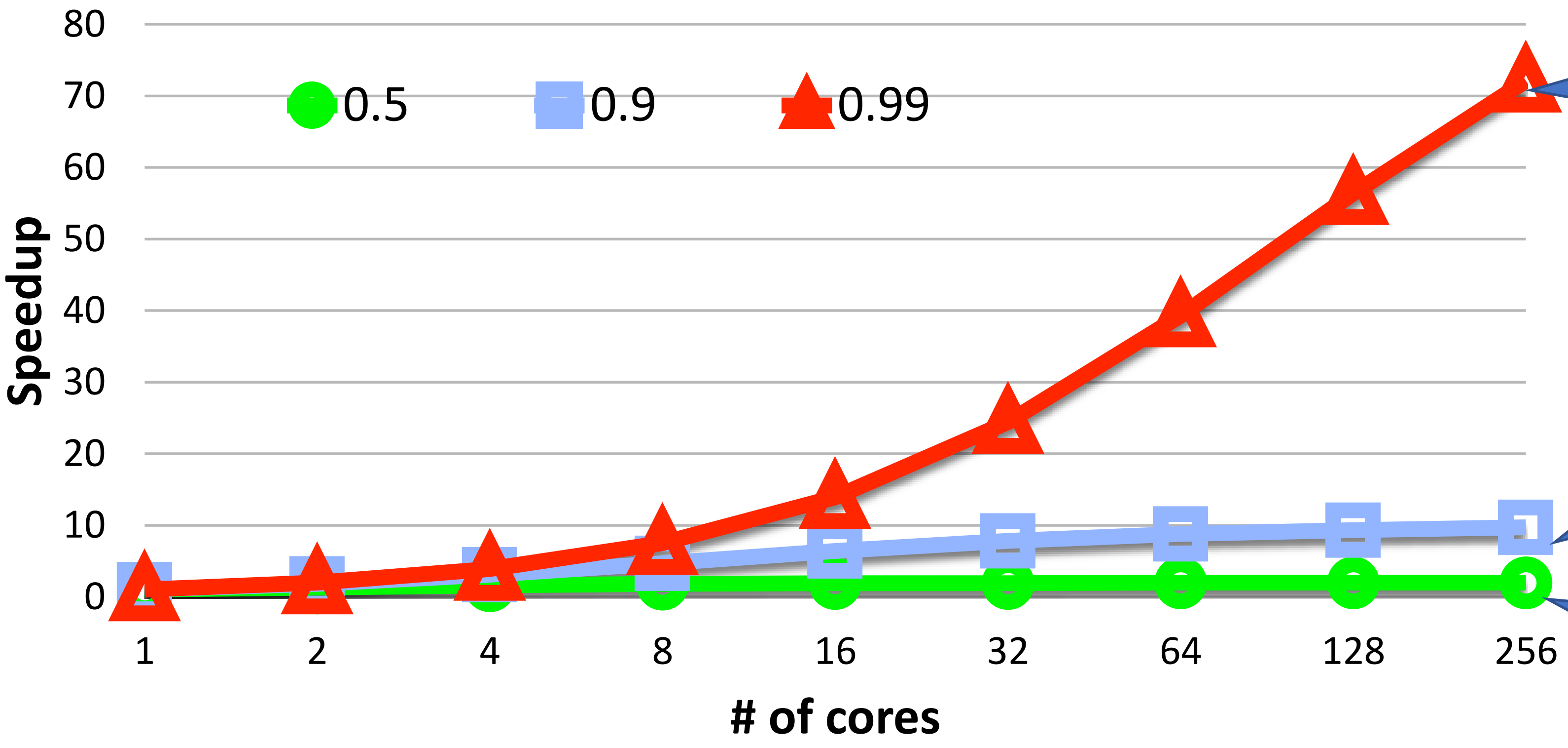
2. Gives an upper bound on parallel speedup

Only fraction F can be concurrently run by *p* processors

- Increasing *p* cannot speed up fraction 1-F

- **Speedup with p processors,** $\boldsymbol{S_p = \frac{1}{1-\mathrm{F} + \frac{F}{p}}}$
- **Upper bound on speedup at** $S_\infty = \frac{1}{\mathbf{1-F}}$
- **Example: If F = 0.67:**
  - Speedup for 4 cores = 2
  - Speedup for ∞ processors = 3

Parallel Speedup

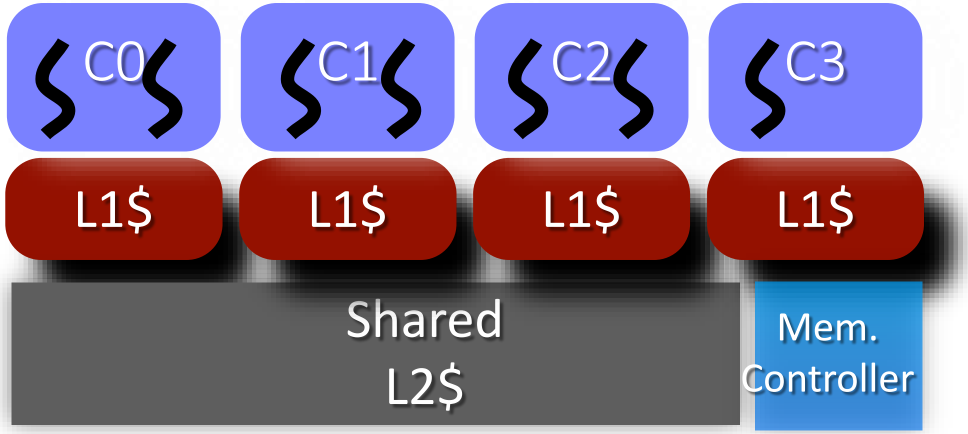


99% Parallel
72x speedup

90% Parallel
9.7x speedup

50% Parallel
1.99x speedup

Amdahl's Law for Multicores [Hill and Marty, 2008]



Multicore Chip partitioned into

- multiple cores (includes L1 cache)
- uncore (Intel terminology for Shared L2 cache, L3)

Resources per-chip bounded

- Area, Power, $, or a combination
- Bound of total N resources per-chip.
- How many cores ? How big is each core ?

Core Types

- **Single-core performance can be improved using same resource: out-of-order execution, multiple pipelines (superscalar processors), larger caches, etc.**
  - Becoming increasingly hard to do in a power-efficient way

- **Small (Wimpy) Core :**
  - Consumes 1 BCE (Base Core Equivalent: measure of core resources)
  - Performance = 1

- **Large Core:**
  - Consumes R BCEs
  - Performance = perf(R)

Large Cores

- **If Perf (R) >= R ; always use large cores**
    - ➢Speeds up everything

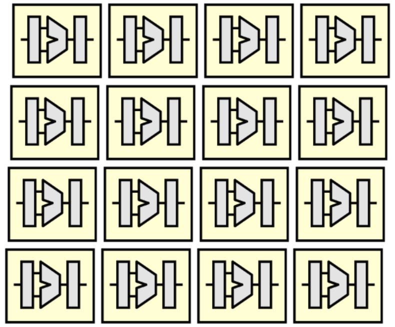
- **Unfortunately, Perf (R) < R typically**

- **Assume Perf (R) = $\sqrt{R}$**
    - ➢Microprocessor examples seem to indicate this is a reasonable assumption

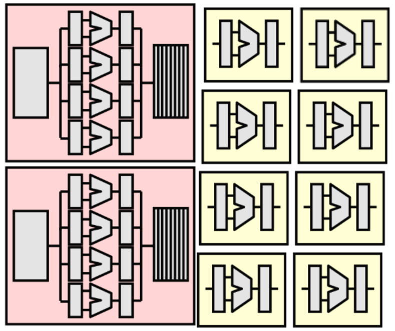
- **How to design a core for specific Perf (R)?**
    - ➢Basic idea: Execute many instructions in parallel

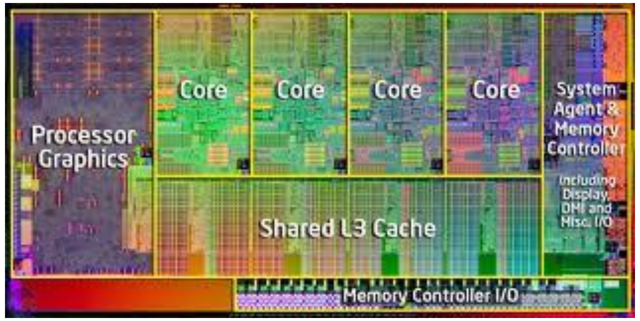
Multicores under consideration

Symmetric

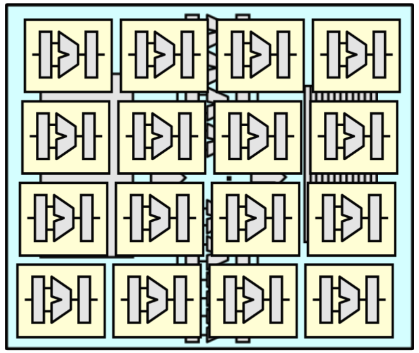


Asymmetric

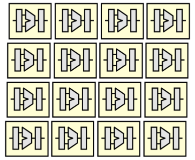




Dynamic (Morphing)



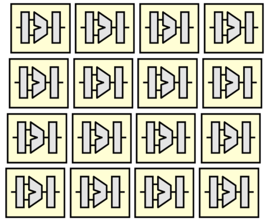
Symmetric Multicores



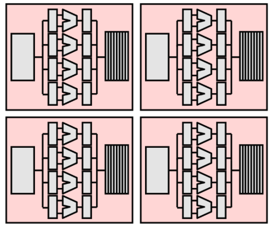
- How many cores ? How big is each core ?
- Chip is bounded to N BCEs
  - Each core has R BCEs
- Number of cores per chip = N/R

- For example, assume N = 16

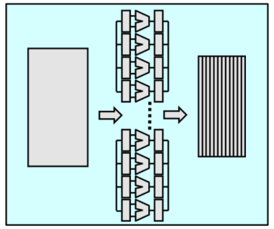
R = 1



R = 4



R = 16



Symmetric Multicore : Performance

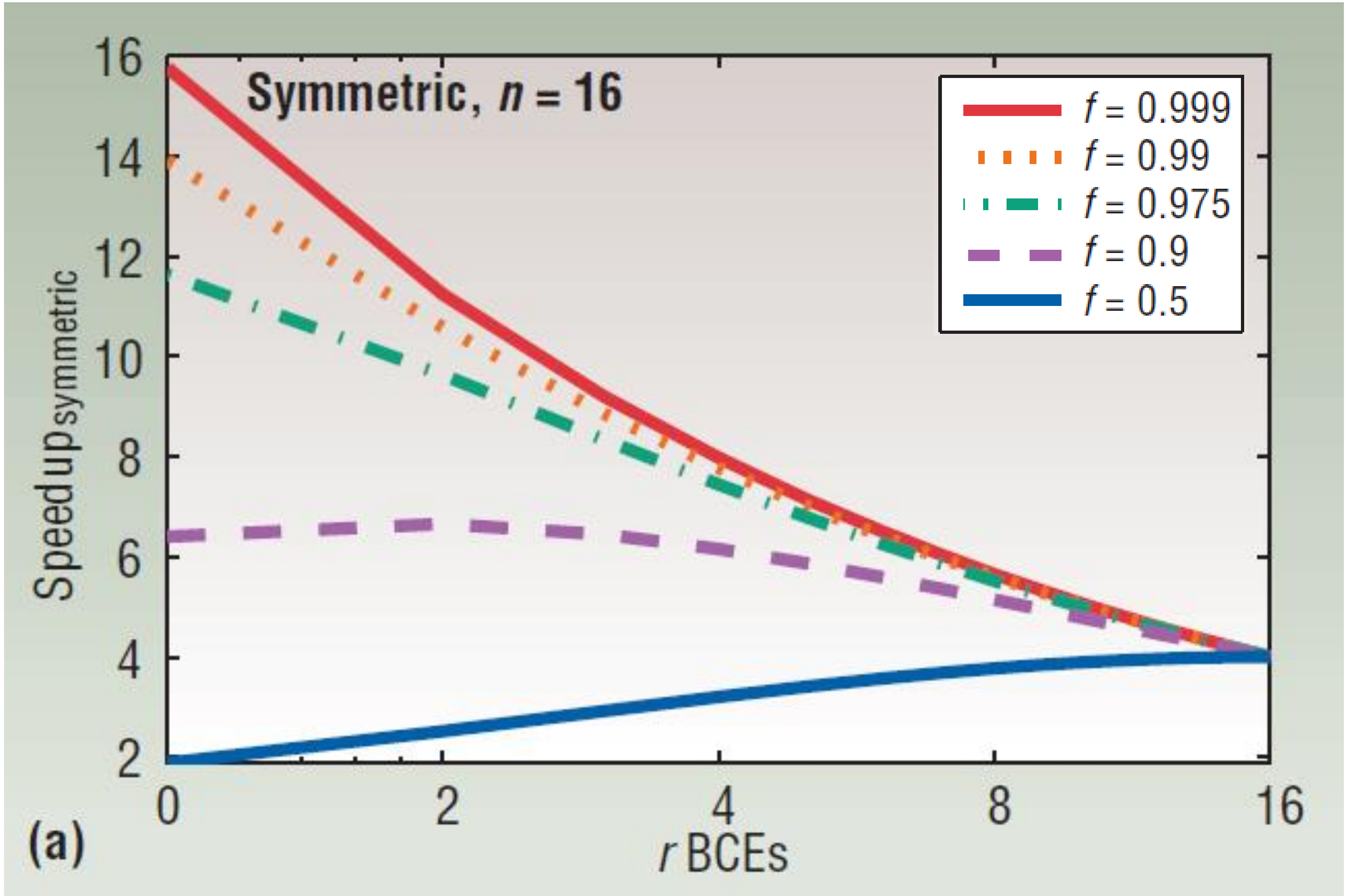
- **Serial Phase (1-F) runs on 1 thread on 1 core**
  - performance α Perf (R)
  - Execution time = (1-F) / Perf (R)
- **Parallel Phase uses all N/R cores. Core @ Perf (R)**
  - Execution time = F / [Perf (R) * N/R]

$$\textit{Speedup (F, N, R)} = \frac{1}{\frac{1-F}{Perf(R)}+\frac{F*R}{Perf(R)*N}}$$

Serial Phase ↑
Perf(R)

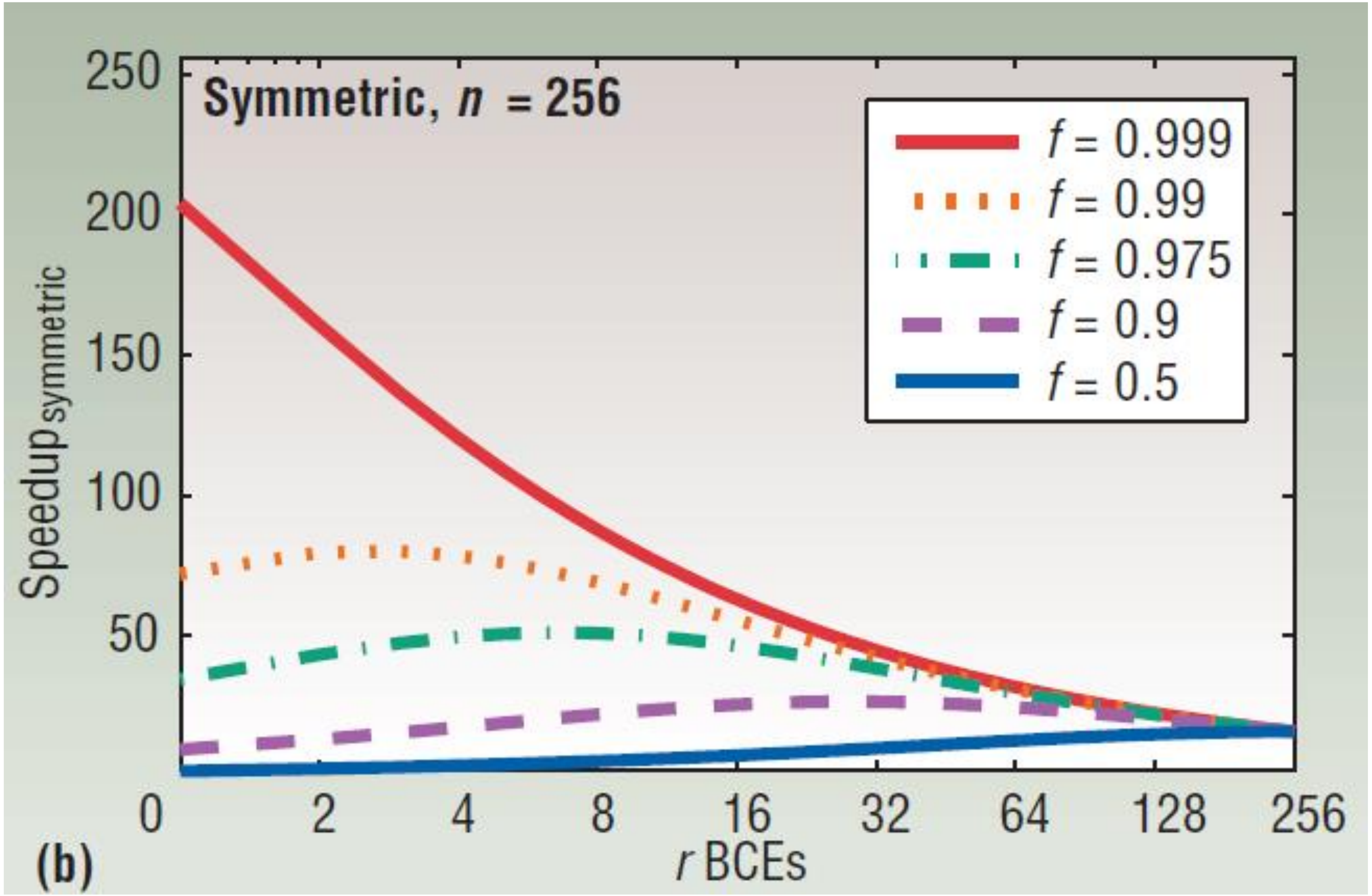
Parallel Phase
More cores!

Symmetric Multicore (Chip = 16 BCEs)



Hill & Marty 2008

Symmetric Multicore (Chip = 256 BCEs)



Hill & Marty 2008

Model-bias towards parallelism

Remember Perf (R) when scaling up CPU: Perf(R) = $\sqrt{R}$

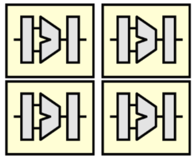
Lets say 1st generation 1 BCE system = 1 BCE

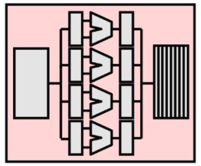
Now consider 2nd gen 4 BCE system

- Four 1BCE cores or One 4BCE core?
- When F=0.999; always pick Four 1CU cores

F=0.999

Speedup ~4





Speedup = 2

- Even parallel fraction not perfectly parallel
  - ➢ Synchronization, Contention, Locks etc.
  - ➢ Need Parallel Software: Perf(R) (depends on application)

Multicore Moore's Law

Since 1970s Technology Moore's Law

- Double transistors every 2 years.
- Should possibly continue....

Microarchitect's Moore's Law

- Double single-thread performance every 2 years
- Stopped due to power wall

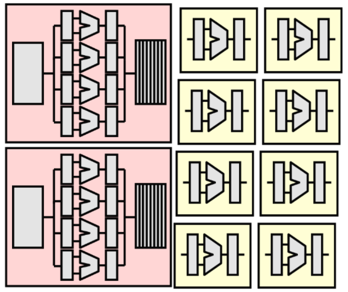
Multicore's Moore's Law

- 2x cores every 2 years
- Need to double software threads every two years
- Need HW to enable 2x threads every two years

Cost-Effective Multicore Computing

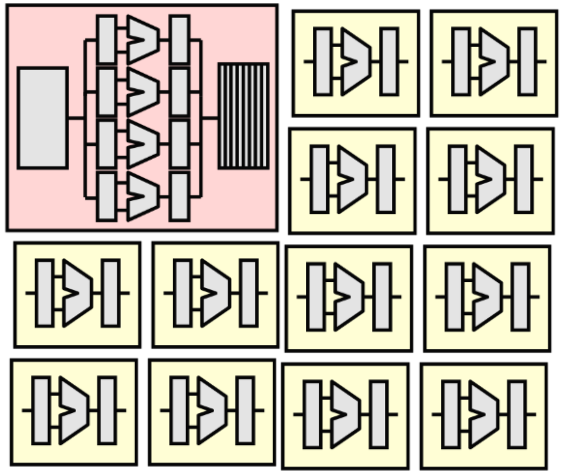
- **Even if Speedup (N cores) < N, a multicore system could still be cost-effective, depending on cost of adding cores**
  - ➢$$$, Power
  - ➢Cost-ratio = Cost (N_{cores}) / Cost (1)
- **If cost is proportional to chip budget, Cost-ratio << N.**
  - ➢Much of multicore cost outside core: Caches, Memory Controller, SSD etc.
- **If cost is proportional to power, cost-ratio can approach N**
- **Multicore computing effective if Cost-ratio < Speedup**
  - ➢Example: 6-core costs $1600; 10-core costs $2000
  - ➢If 10-core speedup >1.25x 6-core, then cost-effective

Asymmetric Multicores



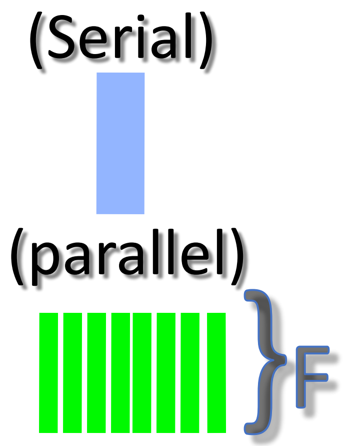
- Enhance some cores to improve performance for serial phase.

- Total chip resources = N BCEs
- Assume two-types of cores on-chip
  - One core = R BCE, N-R 1 BCE cores
  - Total cores = N-R+1



Asymmetric Cores : Performance

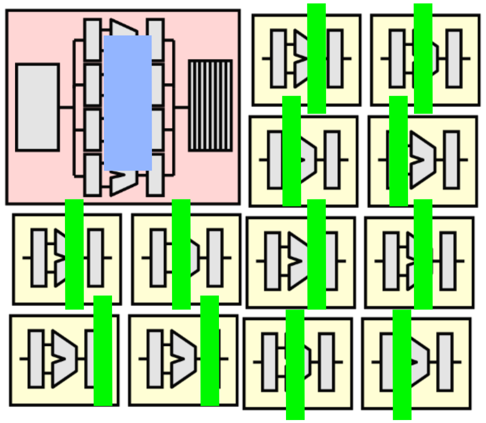
Program Phases



(Serial)

(parallel)

F



Speedups with 1 large core:

Serial Phase = (1-F)/Perf(R)

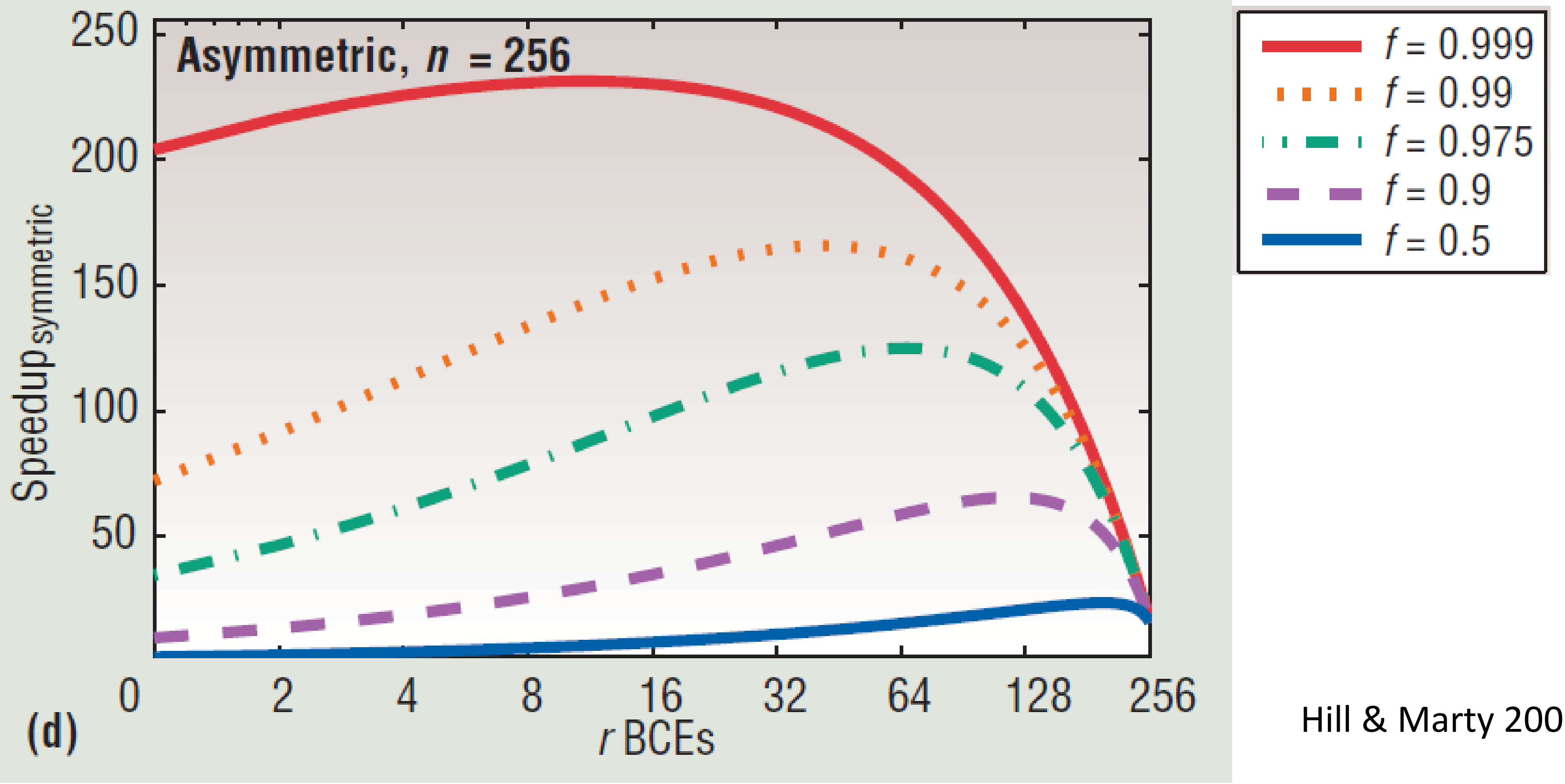
Parallel Phase = F/(Perf(R)+N-R)

$$\textit{Speedup (F, N, R)} = \frac{1}{\frac{1-F}{Perf(R)} + \frac{F}{Perf(R)+N-R}}$$

Asymmetric Multicore (Chip = 256 BCEs)

Asymmetric cores offer great potential

with 1 large core, speedup increases significantly

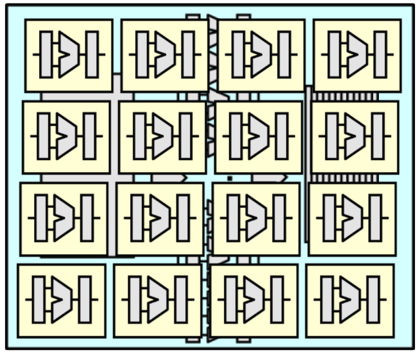


Hill & Marty 2008

Asymmetric Multicores : Challenges

- **Task Management:**
  - ➢How to schedule Computation
  - ➢How to determine which tasks go to large cores vs. small cores
- **Locality:**
  - ➢How to keep data closer to task
  - ➢Also a challenge for symmetric multicore
- **Coordination:**
  - ➢How to synchronize data
  - ➢Harder when cores are not the same

Dynamic (Morphing) Multicores



- **Chip consists of N 1BCE cores**
  - Efficient for parallel phase

- **At runtime glue R 1BCE cores to create R BCE core**
  - Improves performance for serial phase

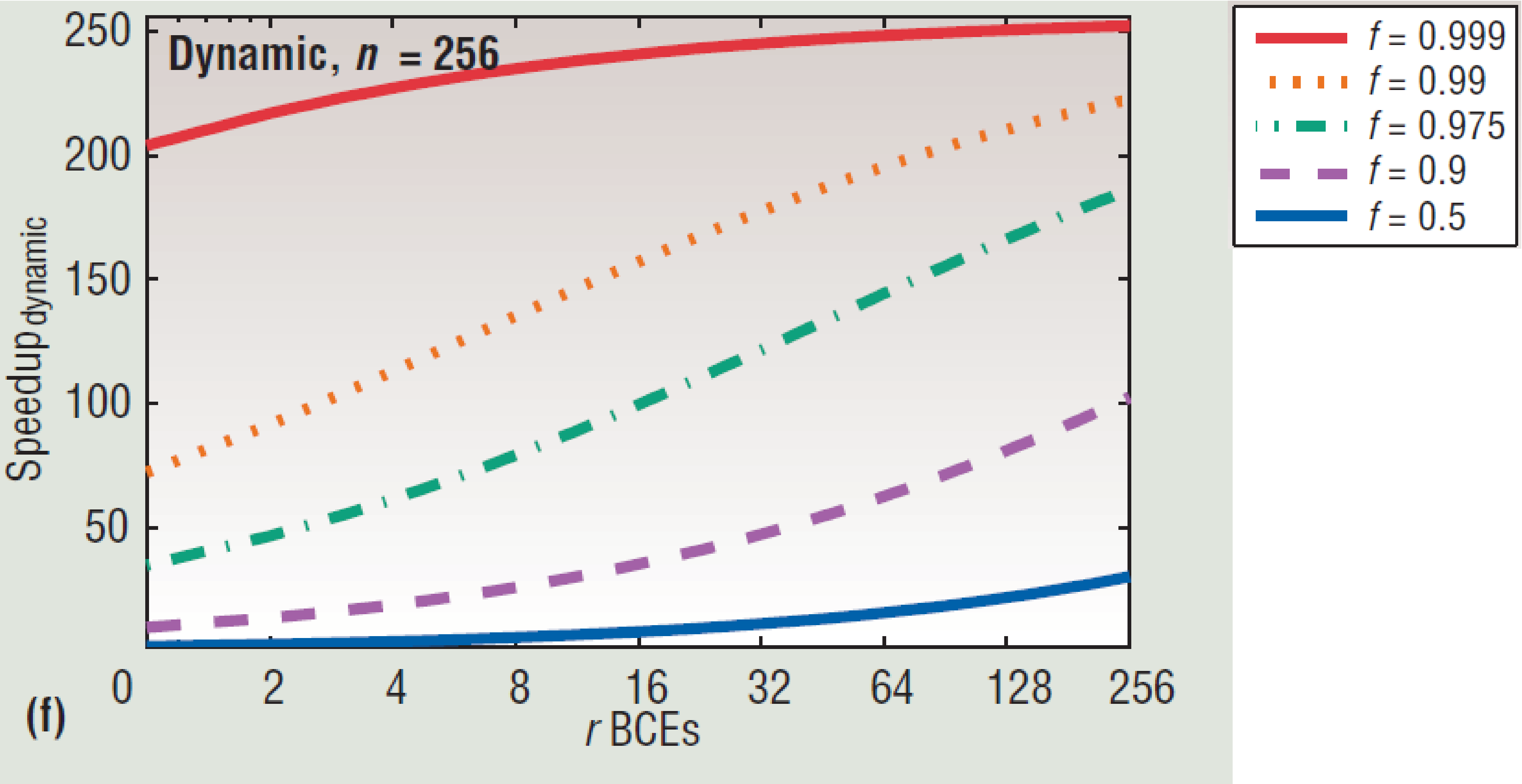
- **How to dynamically glue cores ?**
  - Research still needed

Dynamic Multicores : Performance

- **N 1BCE cores, from which R 1BCE cores glued**
- **Serial phase uses R BCE core at Perf (R)**
  - execution time = (1-F)/Perf(R)
- **Parallel phases uses N cores**
  - execution time = F/N

$$\textit{Speedup}\ (F, N, R) = \frac{1}{\frac{1-F}{Perf(R)} + \frac{F}{N}}$$

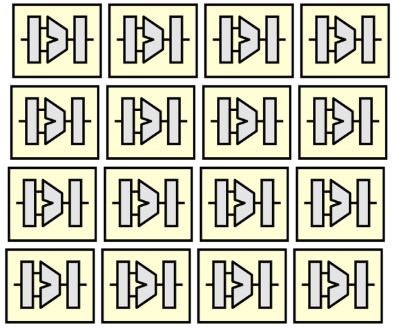
Dynamic Multicore (Chip = 256 BCEs)



Hill & Marty 2008

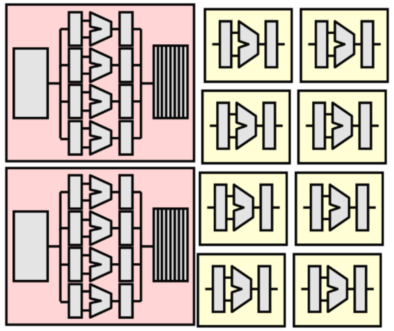
Summary: Multicore Amdahl's Law

Symmetric



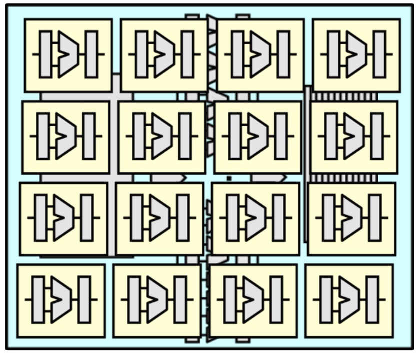
$$\frac{1}{\frac{1-F}{Perf(R)}+\frac{F*R}{Perf(R)*N}}$$

Asymmetric



$$\frac{1}{\frac{1-F}{Perf(R)}+\frac{F}{Perf(R)+N-R}}$$

Dynamic



$$\frac{1}{\frac{1-F}{Perf(R)}+\frac{F}{N}}$$

Shared Memory Multiprocessors & Cache Coherence Protocols

Shared Memory Multiprocessors

- **All processors can access all memory**
- **Processors share memory resources, but can operate independently**
- **One processor’s memory changes are seen by all other processors**
- **Easier to program**
    - ➢Communication through shared memory
    - ➢Synchronization through locks stored in shared memory
- **Need cache coherence in hardware**
- **Need interconnection network between all processors and all memory**
- **Two Types:**
    - ➢Uniform Memory Architectures (UMA): e.g., Symmetric Multiprocessors
    - ➢Non-Uniform Memory Architectures (NUMA): Access & latency to memory is different

Interconnection Networks

- **In a shared memory MP, we need to connect different processors and memory modules**
- **Types of interconnect:**
  - ➢Shared bus
  - ➢Crossbar: Fully connected
  - ➢Ring
  - ➢Mesh
  - ➢2-D Torus
  - ➢Hypercube
- **Number of hops vs. number of links: Compare N processors and M memory modules**

Shared Memory Multiprocessors: Memory Hierarchy

- **Problem: sharing memory means more than one processor can send requests to memory**
  - ➢High memory bandwidth required
- **To avoid sending lots of memory requests, processors use caches to:**
  - ➢Filter out many memory requests
  - ➢Reduce average memory latency
  - ➢Reduce memory bandwidth requirements
- **Typically more than one level of caches is used**
  - ➢L1 caches: Usually Split I & D caches, small and fast
  - ➢L2 caches: Usually on die, composed of SRAM cells
  - ➢L3 caches: On-die or off-die, SRAM or eDRAM cells

Cache Coherence

- **Problem: Using caches means multiple copies of the same memory location may exist**
  - ➢Updates to the same location may lead to bugs
- **Example:**

  Processor 1 reads A

  Processor 2 reads A

  Processor 1 writes to A

Now, processor 2's cache contains stale data

- **Cache coherence need to be implemented in hardware using a cache coherence protocol**

Conditions for Cache Coherence

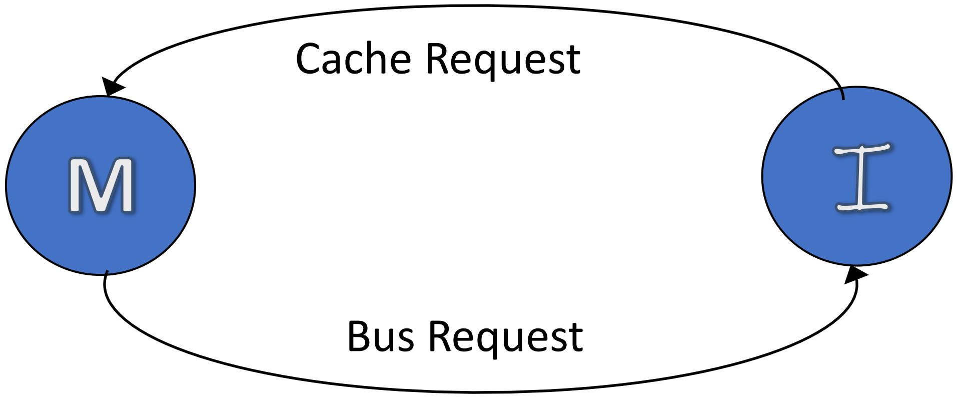
- **Program Order.** A read by processor P to location A that follows a write by P to A, with no writes to A by another processor in between, should always return the value of A written by P
- **Coherent View of Memory.** A read by processor P1 to location A that follows a write by another processor P2 to location A should return the written value by P2 if:
  - The read and write are sufficiently separated in time
  - No other writes to A by another processor occur between the read and the write
- **Write Serialization.** Writes to the same location are serialized: Two writes to the same location by any two processors are seen in the same order by all processors

Cache Coherence Protocol Classification

- **Cache coherence defines behavior of reads and writes to the same memory location**
- **Compared to: Memory consistency models define the behavior of reads and writes with respect to accesses to other memory locations**
- **Two main types of cache coherence protocols:**
    - ➢Snooping
        - ❑Caches keep track of the sharing status of all blocks
        - ❑No centralized state is kept
        - ❑Cache controllers snoop shared interconnect to know when a requested block exists in the cache
    - ➢Directory
        - ❑Sharing status of any block in memory is kept in one location

Very Simple Coherence Protocol

- **MI protocol**
  - Two states: M (Modified) and I (Invalid)
  - Only one cache contains a copy of a certain memory location
  - When another cache requests a block, the cache currently containing the block invalidates it
  - Protocol limits sharing and degrades performance



- **Optimization: MSI protocol allows read sharing**

Announcements

- **Reading Assignments**
  - ➢Lawrence Livermore National Lab, “Introduction to Parallel Computing,” https://hpc.llnl.gov/training/tutorials/introduction-parallel-computing-tutorial (Skim)
  - ➢M. Hill and M. Marty, “Amdahl Law in the Multicore Era,” IEEE Computer 2008 (Read)
  - ➢ARCH Chapter 5.1, 5.2 (Read)
- **Exam 2 Wednesday (Same logistics as Exam 1)**
- **Project proposals due Friday**
- **Assignment 3 due Monday**