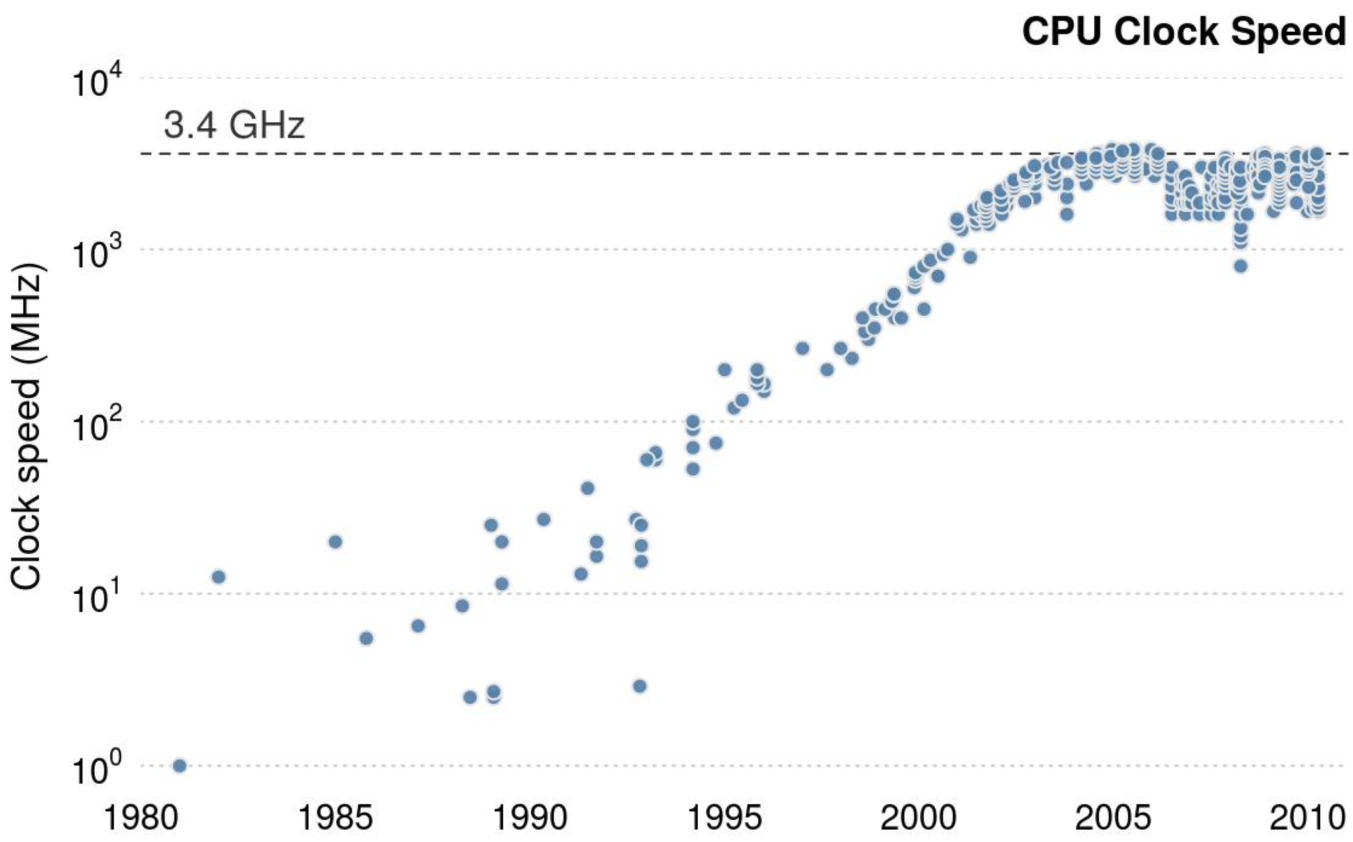
PBA Software Development
Parallel Programming

Parallel Programming with C#.NET

Introduction

Clock Speed Stagnation



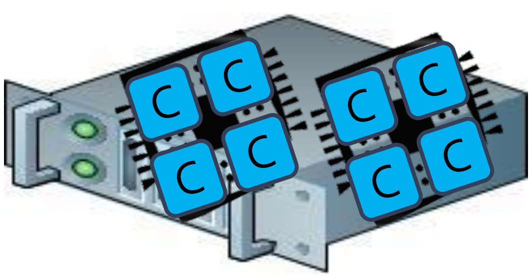
Clock Speed not increasing?

- Moore's law is still alive and well
  - transistor counts still doubling every 18 months…
  - memory & disk storage still doubling…

- Why have clock speeds *stopped* doubling?
  - Heating problems in the CPU due to nescecarry power increase.

Performance increase

- The Multi-core era has started.
- If we want better performance, we must take advantage of the multiple cores.



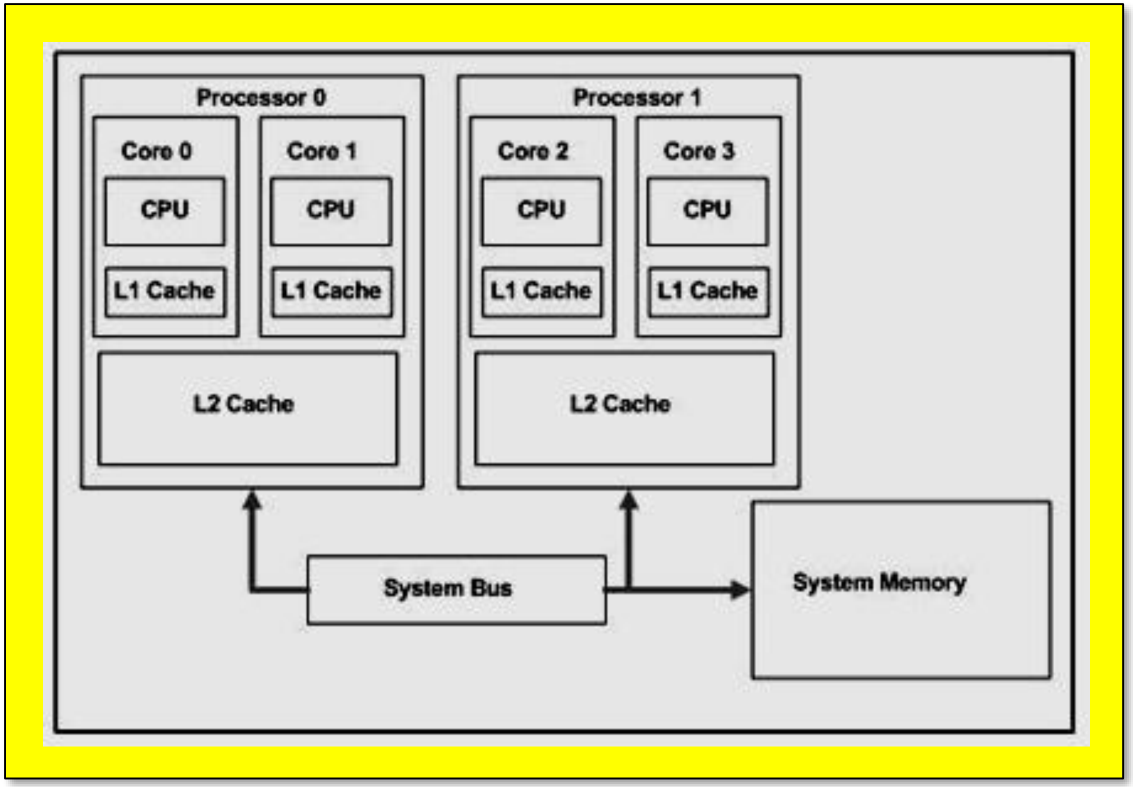
8 x 2GHz cores have the computing equivalent of 1 x 16GHz core, yet require O(logN) power increase

Multicore implications

- Better performance running *multiple* programs
  - OS will run each program on a *different* core
- What about speeding up a *single* program?
  - Can hardware automatically spread instruction stream across the cores?
  - Can software – e.g. compiler – automatically split program to run across multiple cores?
- The answer is "NO"!
- To make a ***single*** program run faster, we need to ***explicitly*** program **for multicore** execution.

Multicore Architecture

- Key features:
  - Sockets
  - Cores
  - Caches
  - Cache coherence
  - RAM



Hyper-Threading

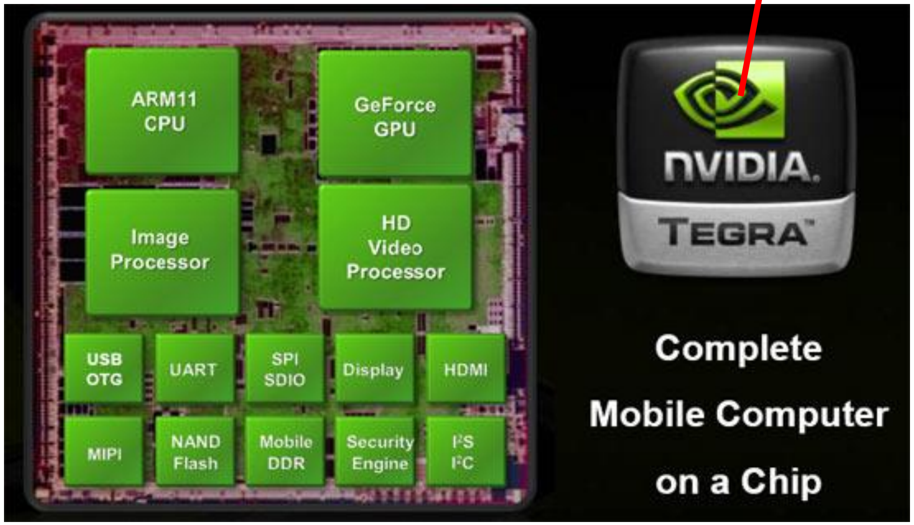
- The cores may use hyper-threading, so more threads can be executed at the same time simultaneously by the core.

- **Four physical cores** in the CPU may seem as **eight logical cores** to the Operating System due to hyper-threading.

Manycore?

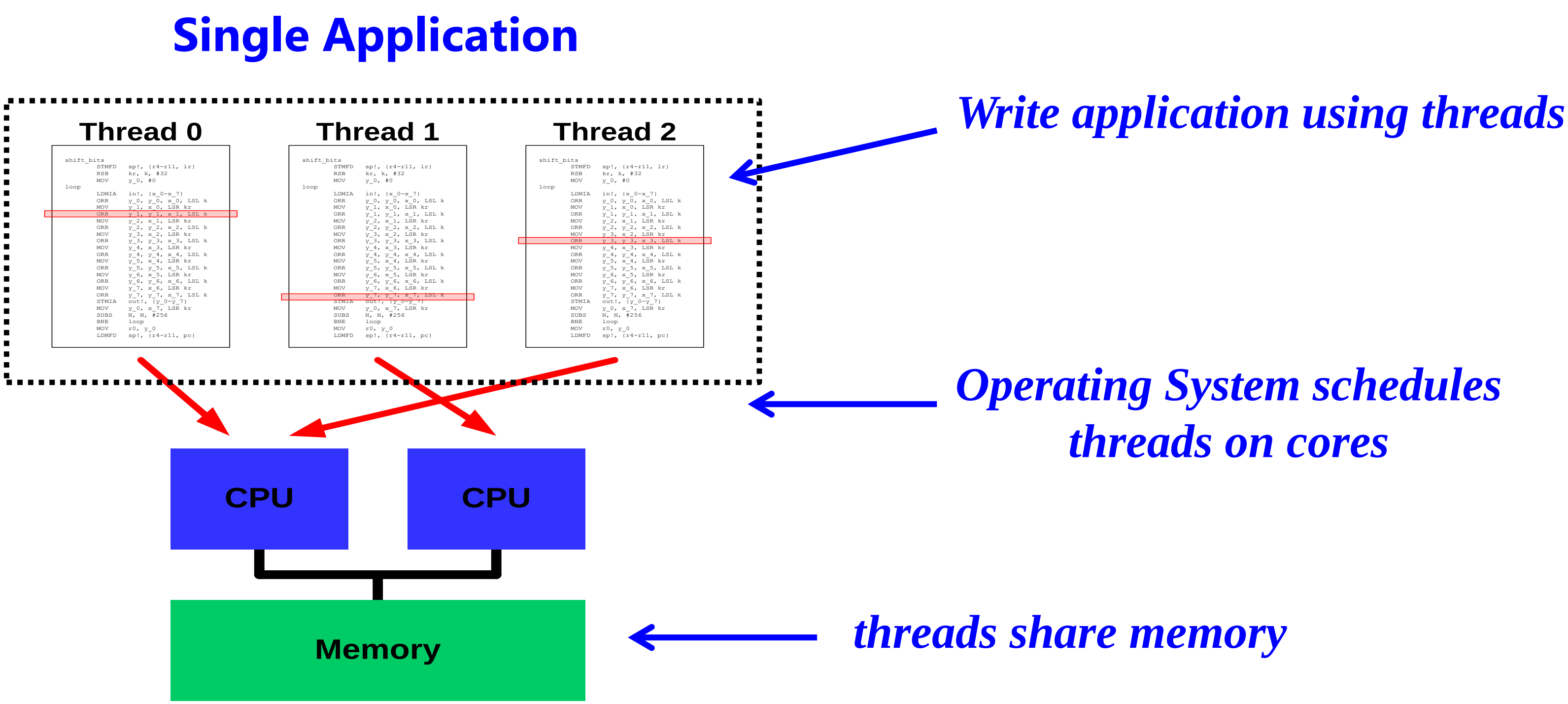
- Manycore implies *different* types of cores
  - General-purpose CPU cores
  - Graphics processing (GPU) cores
  - Video processing cores
  - Network processing cores
  - And more…

Single "APU"



Basic Multicore Programming Model

- Basic Programming model is Thread-based:



Amdahl's Law

- The execution time for a parallel processed algorithm is dependent on the percentage of the algorithm, that cannot be parallelized.
- If
  - ***B*** is the **percentage** of the code being **serial** (non-parallel).
  - ***T(n)*** is the executing time using ***n*** **parallel threads**.
- Then

$$T(N) = T(1) \cdot \left(B + \frac{1}{n} \cdot (1 - B) \right)$$

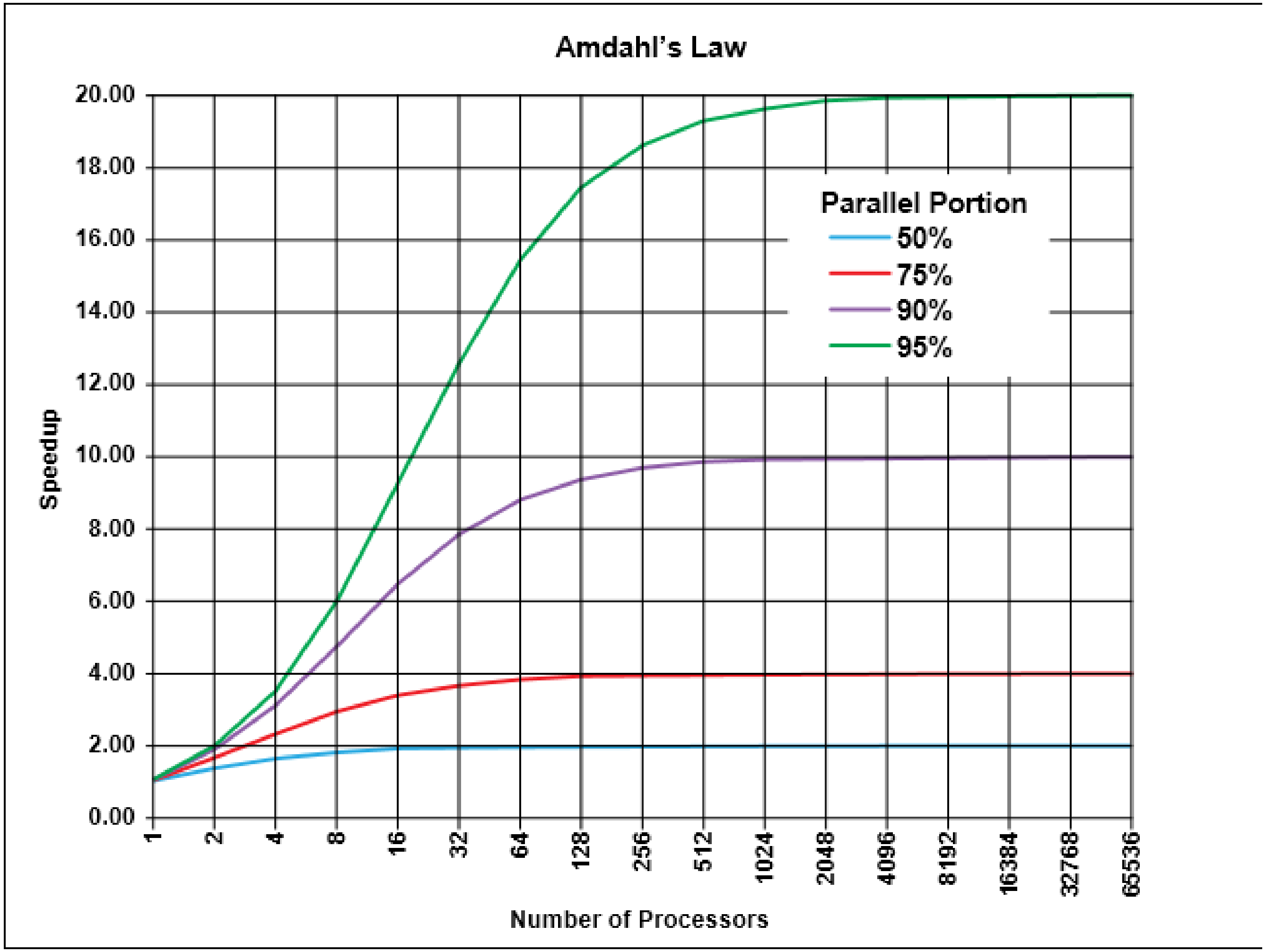
Amdahl's Law

- The theoretically defined maximum speed increase by parallelizing is

$$S(n) = \frac{T(1)}{T(n)} = \frac{1}{B + \frac{1}{n} \cdot (1 - B)}$$

Or for a large number of parallel threads

$$S(n) \rightarrow \frac{1}{B}\ , for\ n \rightarrow\ \infty$$


Amdahl's Law
Speedup
20.00
18.00
16.00
14.00
12.00
10.00
8.00
6.00
4.00
2.00
0.00
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536
Number of Processors
Parallel Portion
50%
75%
90%
95%

Potential Parallel

- Never design parallel algorithms for a specific number of cores.
- Even is the number of cores in known in advance we cannot be sure that the cores is available when needed.
- We need to program in a way, so the algorithm benefits from the **actual number of cores available at execution time**.
- **Future Proof** implementation if more cores will be available in future architectures.

Microsoft TPL

- Task Parallel Library
  - Microsoft-specific concurrency API for C#
  - First appeared in Visual Studio 2010
  - Library-only approach: compiler not involved

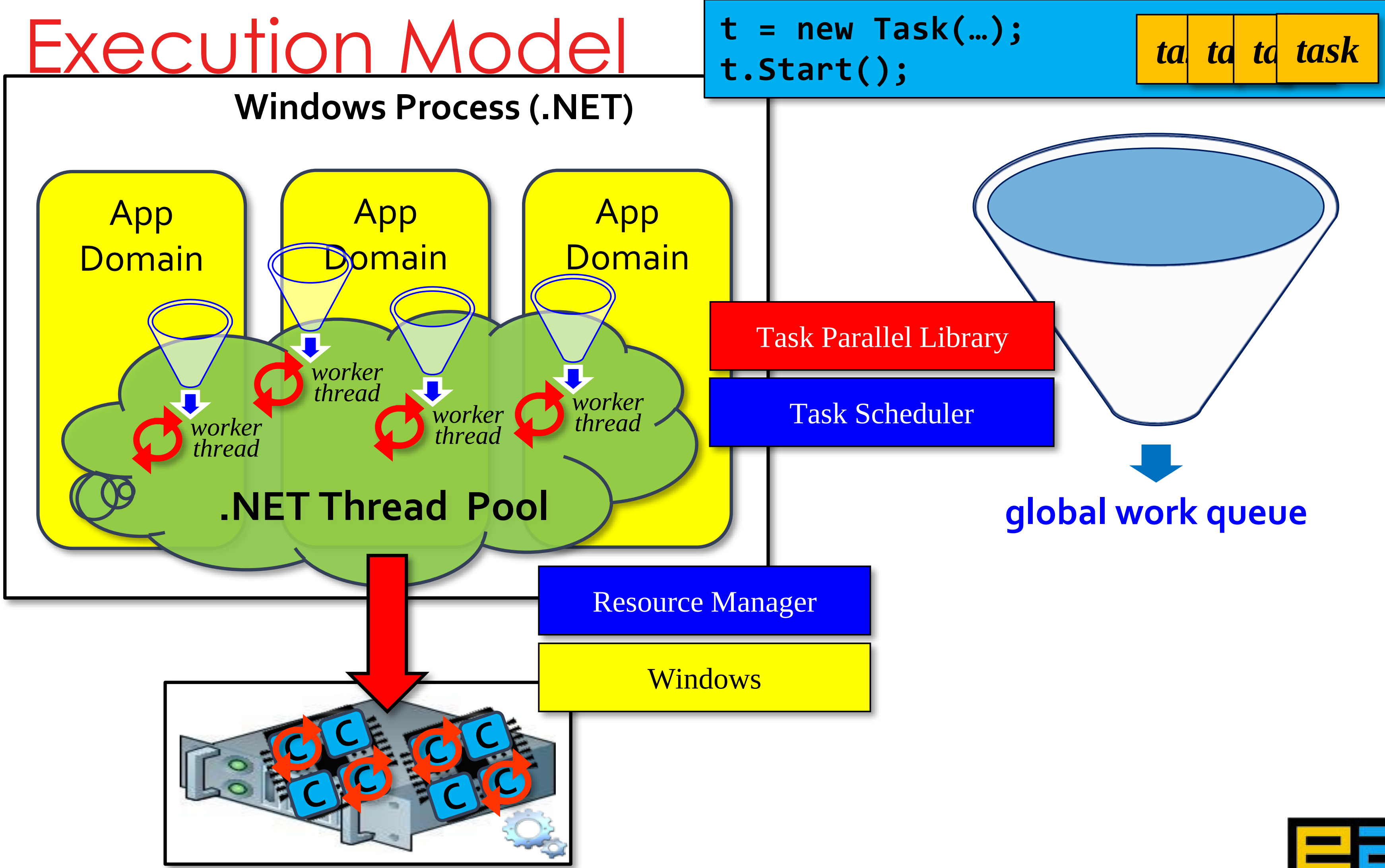
- Support for:
  - data parallelism
  - task parallelism
  - asynchronous operations
  - parallel data structures

Tasks

- A Task defines a unit of work which then can be executed by a thread on a core.
- Tasks decouples the actual job to be done from the physical threads executing on the cores.
- The Operating system has multiple threads in a thread-pool designated for each core.
- When executing a task-based algorithm, the task objects are created in a task-pool, and then assigned for an idle thread to execute by a core.

Execution Model

```
t = new Task(…);
t.Start();
```



Overview of TPL

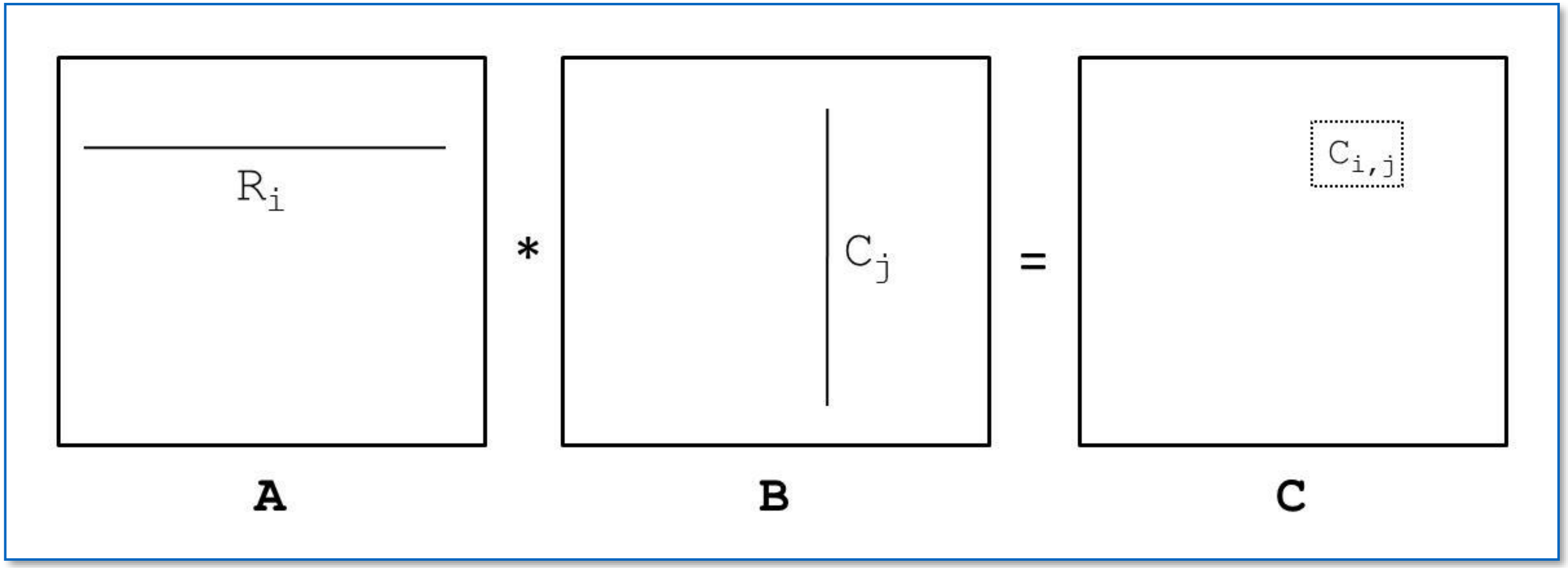
- **Waiting for tasks to finish:**
  - `.Wait`
  - `Task.WaitAll`
  - `Task.WaitAny`
- **Harvesting results:**
  - `Task<TResult>`
  - `.Result`
- **Composition:**
  - `.ContinueWith`

- **Exception handling**
- **Cancelling**
- **Higher-level constructs:**
  - `Parallel.For`
  - `Parallel.ForEach`
  - `Parallel.Invoke`
- **Data structures:**
  - `ConcurrentQueue`
  - `ConcurrentStack`
  - `BlockingCollection`
  - `...`

Demo

$$C[i,j] = \sum_{k=0}^{n-1} (A[i,k] \cdot B[k,j])$$

- Matrix-multiplication



Demo

- Matrix-multiplication (Naive implementation)

```
for (int i = 0; i < N; i++)
{
    for (int j = 0; j < N; j++)
    {
        C[i][j] = 0.0;

        for (int k = 0; k < N; k++)
        {
            C[i][j] += (A[i][k] * B[k][j]);
        }
    }
}
```

Cashing of data by the processor

- Caching (L1 and L2) in the processor is done in **row-major order**, which means that data from the same row in the matrix is read into the CPU consecutively.
- **Problem:**
  In the naive implementation the B matrix was accessed column-wise, which leads to heavy swapping in the cash.
- **Solution**:
  Read both matrices in row-major order.

Loop interchange

- Cache-friendly matrix-multiplication

```
// Initialize result matrix for cache-friendly multiply:
for (int i = 0; i < Rows; i++)
    for (int j = 0; j < Cols; j++)
        C[i][j] = 0.0;

// Multiply:

for (int i = 0; i < Rows; i++)
{
    for (int k = 0; k < Intermediate; k++)
    {
        for (int j = 0; j < Cols; j++)
            C[i][j] += (A[i][k] * B[k][j]);
    }
}
```

Task parallel version

```
// Initialize result for cache-friendly multiply:
for (int i = 0; i < Rows; i++)
	for (int j = 0; j < Cols; j++)
		C[i][j] = 0.0;

// Multiply:
//
//for (int i = 0; i < Rows; i++)
Parallel.For(0, Rows, (i) =>
{
	for (int k = 0; k < Intermediate; k++)
	{
		for (int j = 0; j < Cols; j++)
			C[i][j] += (A[i][k] * B[k][j]);
	}
});
```

Using Lambda-expression

Parallel Design Patterns

Application characteristic	Relevant pattern
Do you have sequential loops where there's no communication among the steps of each iteration?	The Parallel Loop pattern (Chapter 2). Parallel loops apply an independent operation to multiple inputs simultaneously.
Do you have distinct operations with well-defined control dependencies? Are these operations largely free of serializing dependencies?	The Parallel Task pattern (Chapter 3) Parallel tasks allow you to establish parallel control flow in the style of fork and join.
Do you need to summarize data by applying some kind of combination operator? Do you have loops with steps that are not fully independent?	The Parallel Aggregation pattern (Chapter 4) Parallel aggregation introduces special steps in the algorithm for merging partial results. This pattern expresses a reduction operation and includes map/reduce as one of its variations.
Does the ordering of steps in your algorithm depend on data flow constraints?	The Futures pattern (Chapter 5) Futures make the data flow dependencies between tasks explicit. This pattern is also referred to as the Task Graph pattern.
Does your algorithm divide the problem domain dynamically during the run? Do you operate on recursive data structures such as graphs?	The Dynamic Task Parallelism pattern (Chapter 6) This pattern takes a divide-and-conquer approach and spawns new tasks on demand.
Does your application perform a sequence of operations repetitively? Does the input data have streaming characteristics? Does the order of processing matter?	The Pipelines pattern (Chapter 7) Pipelines consist of components that are connected by queues, in the style of producers and consumers. All the components run in parallel even though the order of inputs is respected.