Concurrency Opportunity Recognition: K-means Algorithm

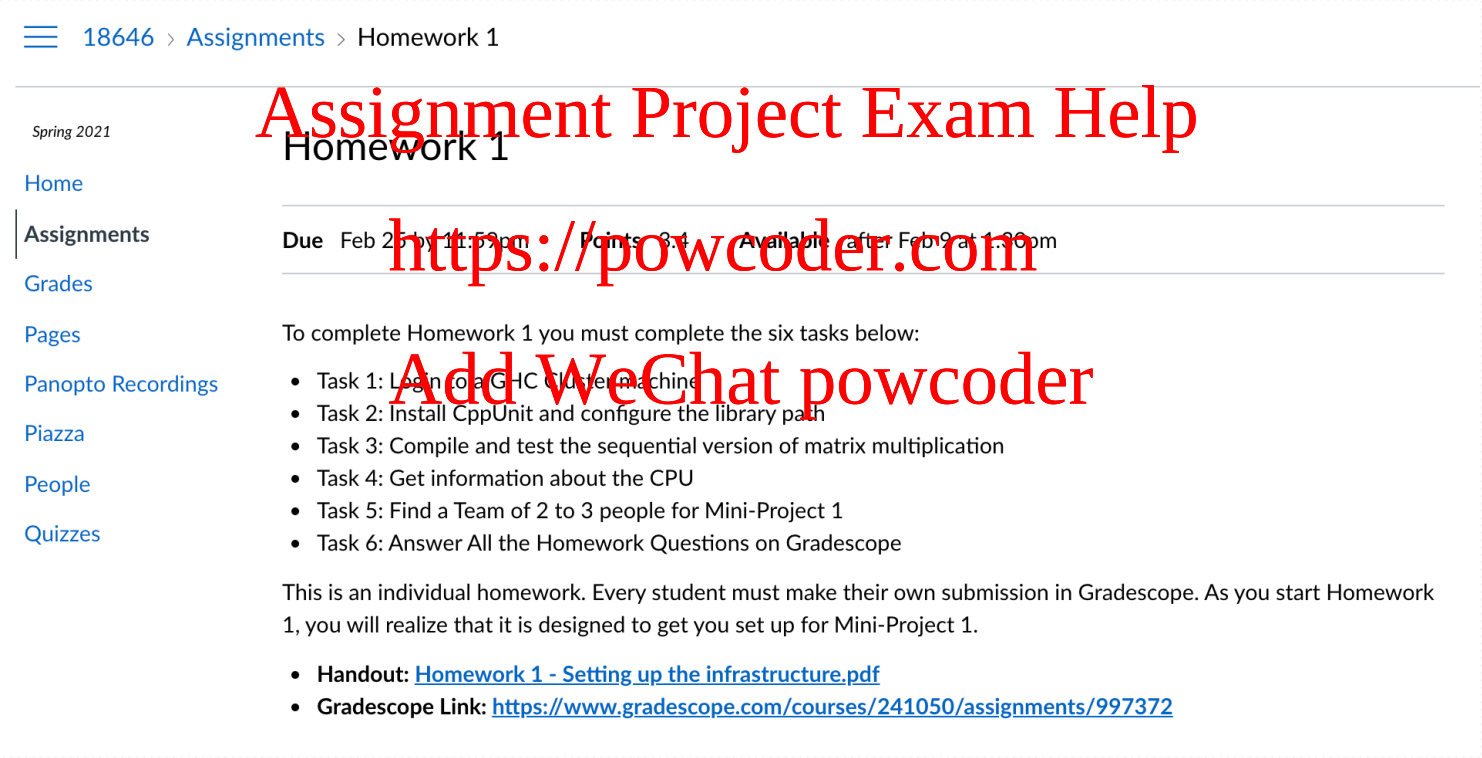
Carnegie Mellon University

Ian Lane

18646 – Week 2

Introduction and Multicore Programming		
Tue, Feb 2	Introduction to How to Write Fast Code II	video and slides
Thurs, Feb 4	Advanced Parallel Hardware Architectures	video and slides
Tue, Feb 9	Concurrency Opportunity Recognition	video and slides Homework 1 Released
Thur, Feb 11	Application Design for Multicore Programming	
Tue, Feb 16	High-Performance Application Optimization On CPUs	**Mini-Project 1 Released**
Thur, Feb 18	Overview of Mini-Project 1	
Tue, Feb 23	No Class	
Thur, Feb 25	Performance Analysis: Roofline Model	**Homework 1 Due**

Homework 1



18646 › Assignments › Homework 1

Spring 2021

Home

Assignments

Grades

Pages

Panopto Recordings

Piazza

People

Quizzes

Homework 1

Due Feb 25 by 11:59pm **Points** 3.4 **Available after** Feb 9 at 1:30pm

To complete Homework 1 you must complete the six tasks below:

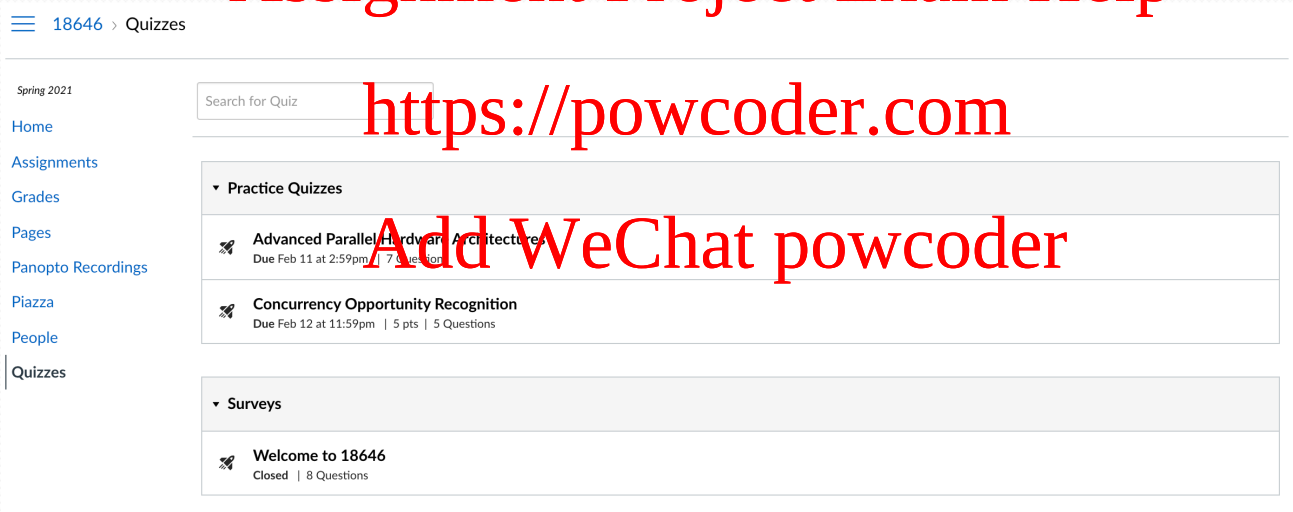
- Task 1: Login to a GHC cluster machine
- Task 2: Install CppUnit and configure the library path
- Task 3: Compile and test the sequential version of matrix multiplication
- Task 4: Get information about the CPU
- Task 5: Find a Team of 2 to 3 people for Mini-Project 1
- Task 6: Answer All the Homework Questions on Gradescope

This is an individual homework. Every student must make their own submission in Gradescope. As you start Homework 1, you will realize that it is designed to get you set up for Mini-Project 1.

- **Handout:** Homework 1 - Setting up the infrastructure.pdf
- **Gradescope Link:** https://www.gradescope.com/courses/241050/assignments/997372

Lecture Questions

https://canvas.cmu.edu/courses/21510/quizzes/55580



Outline – Follow on from 2/4

- Landscape of Computing Platforms
- **Hardware Architectures**
  - Multicore vs Manycore
  - Instruction level parallelism
  - SIMD
  - Simultaneous multithreading
  - **Memory hierarchy**
  - System hierarchy
- How to Write Fast Code?

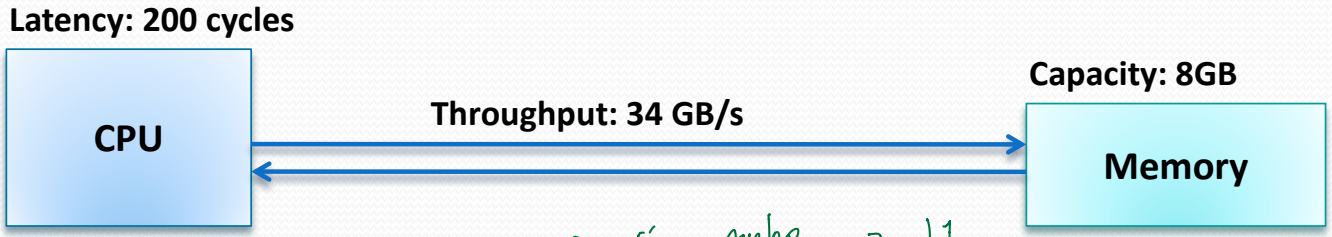
(5) Memory Hierarchy

- **Cache**:
  - A piece of fast memory close to the compute modules in a microprocessor
  - Allows faster access to a limited amount of data
  - Physical properties of wires and transistors determines trade-off between **cache capacity** and **access throughput/latency**

Capacity: Size, e.g. number of bytes of data

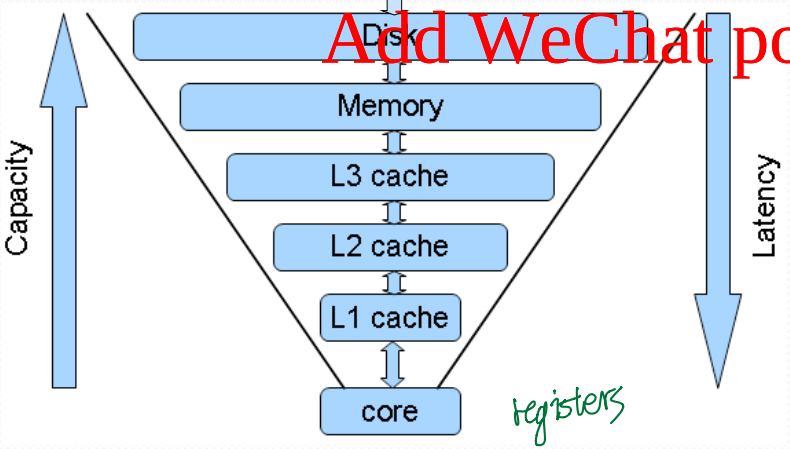
Latency: From start to finish, in units of time, e.g. CPU clock cycles

Throughput: Tasks accomplished per unit time, e.g. GB/s



(5) Memory Hierarchy

- **Cache**:
  - A piece of fast memory close to the compute modules in a microprocessor
  - Allows faster access to a limited amount of data
  - Physical properties of wires and transistors determines trade-off between **cache capacity** and **access throughput/latency**



(Capacity), (throughput), (latency)

Exa-Bytes, 0.1 GB/s, ~1-10B cycles

3GB, 6 GB/s, ~1M cycles

16GB, 34.08 GB/s, ~200 cycles

8 MB, 108.8 GB/s, 26-31 cycles

256 KB, 108.8 GB/s, 12 cycles

32 KB Data Cache, 163.2 GB/s, 4-6 cycles

Intel Core2 – 2600k @ 3.4GHz
(viewed from one core)

Working with a Memory Hierarchy

- **Writing fast code → get data from the fastest (closest) level**
- When requesting data from a level in memory hierarchy of limited size, there are two outcomes
  - **Hit**: Data is available in the level
  - **Miss**: Data is missing from the level
- **For fast code → minimize misses, maximize hits**
- What **application behavior** can a memory hierarchy help with?

locality

Working with a Memory Hierarchy

- **Principle of Locality**
  - The phenomenon of the same value or related storage locations being frequently accessed, usually amenable to performance optimization

- **Temporal Locality**
  - Reuse of specific data and/or resources within relatively small time durations

- **Spatial Locality**
  - Use of data elements within relatively close storage locations

$$A = \begin{pmatrix} 4 & 5 & -5 \\ -1 & -4 & -2 \\ -3 & 1 & 5 \\ 2 & 1 & 4 \end{pmatrix} \qquad B = \begin{pmatrix} -4 & 4 & -1 & 3 & -4 \\ -1 & -5 & -5 & 4 & -5 \\ 1 & 1 & 0 & -1 & 1 \end{pmatrix}$$

$$AB = \begin{pmatrix} -16 & -4 & -29 & 37 & -46 \\ 10 & 18 & 21 & -17 & 22 \\ 6 & -22 & -2 & -10 & 12 \\ -13 & -1 & -7 & 6 & -9 \end{pmatrix}$$

$$(-1)(3) + (-4)(4) + (-2)(-1)$$
$$-3 + -16 + 2$$
$$-17$$

When would you get a miss?

- Three types of misses in a memory hierarchy
  - **Compulsory** misses: caused by the first reference
  - **Capacity** misses: due to the finite size of the memory hierarchy
  - **Conflict** misses: due to policy of replacement, potentially avoidable

Exploiting Benefits of Mem Hierarchy

- Motivation:
  - When application exhibit data locality, cache/memory provides faster access to frequently used data
- Advantages:
  - Allows faster access to data for computation
  - Caches are managed storage: transparent to the end-user for functional purposes
  - Lower the energy consumption when getting a “cache-hits”
- Disadvantages:
  - When using multiple threads share a cache, they will compete for cache space

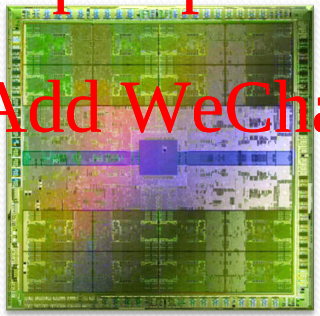
Outline – Follow on from 2/4

- Landscape of Computing Platforms
- **Hardware Architectures**
  - Multicore vs Manycore
  - Instruction level parallelism
  - SIMD
  - Simultaneous multithreading
  - Memory hierarchy
  - **System hierarchy**
- How to Write Fast Code?

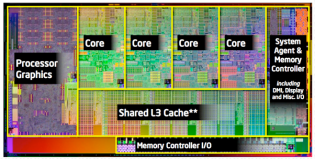
(6) System Architecture

- A Journey through the opportunities to write ***fast code*** with:
  - Multicore
  - Manycore
  - The Cloud

Cloud

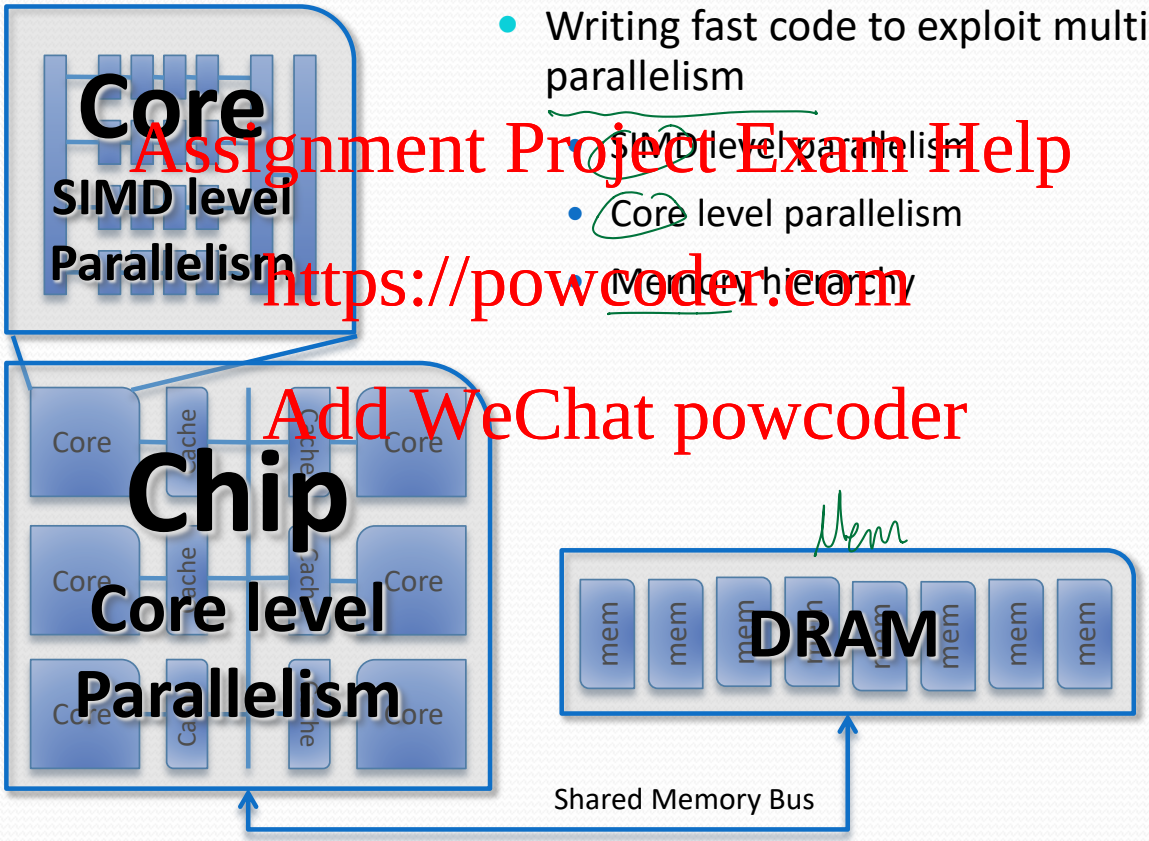


Manycore



Multicore

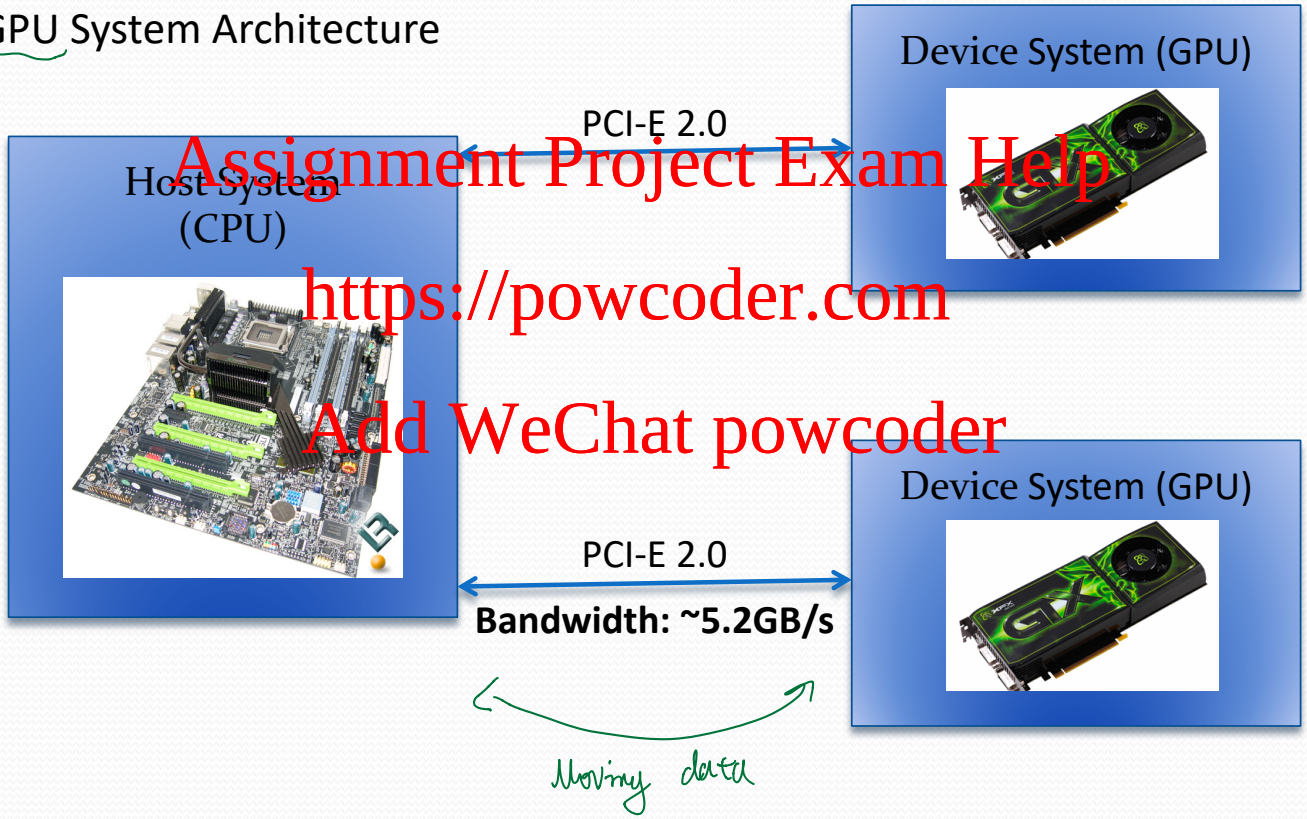
Section 1 - Multicore



- Writing fast code to exploit multicore parallelism
  - SIMD level parallelism
  - Core level parallelism
  - Memory hierarchy

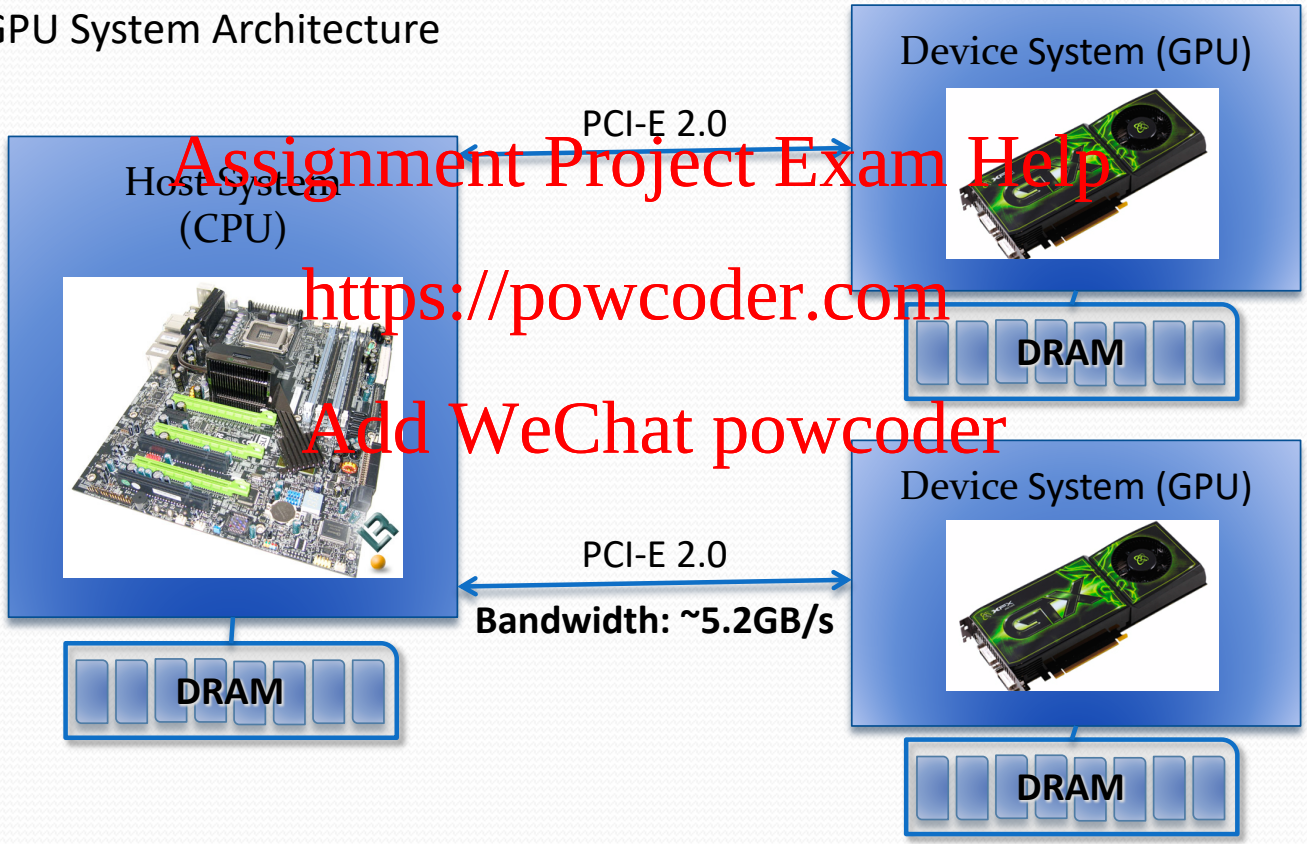
Section 2 - Manycore

- GPU System Architecture

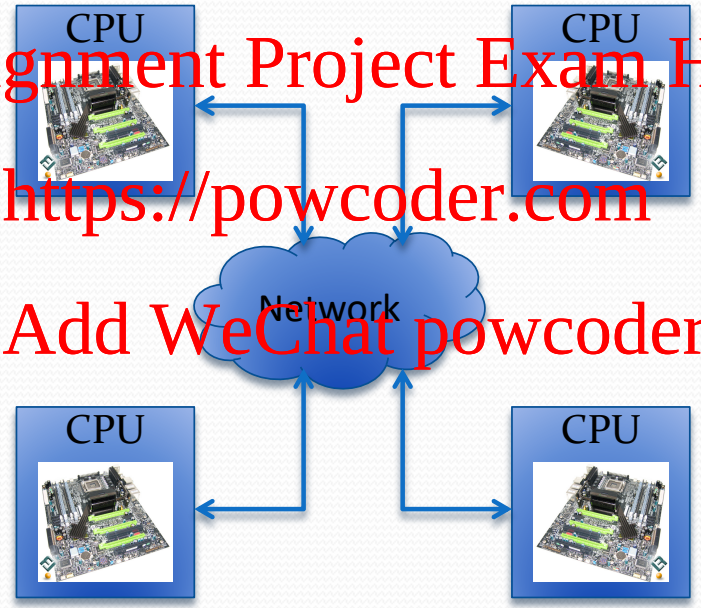


Section 2 - Manycore

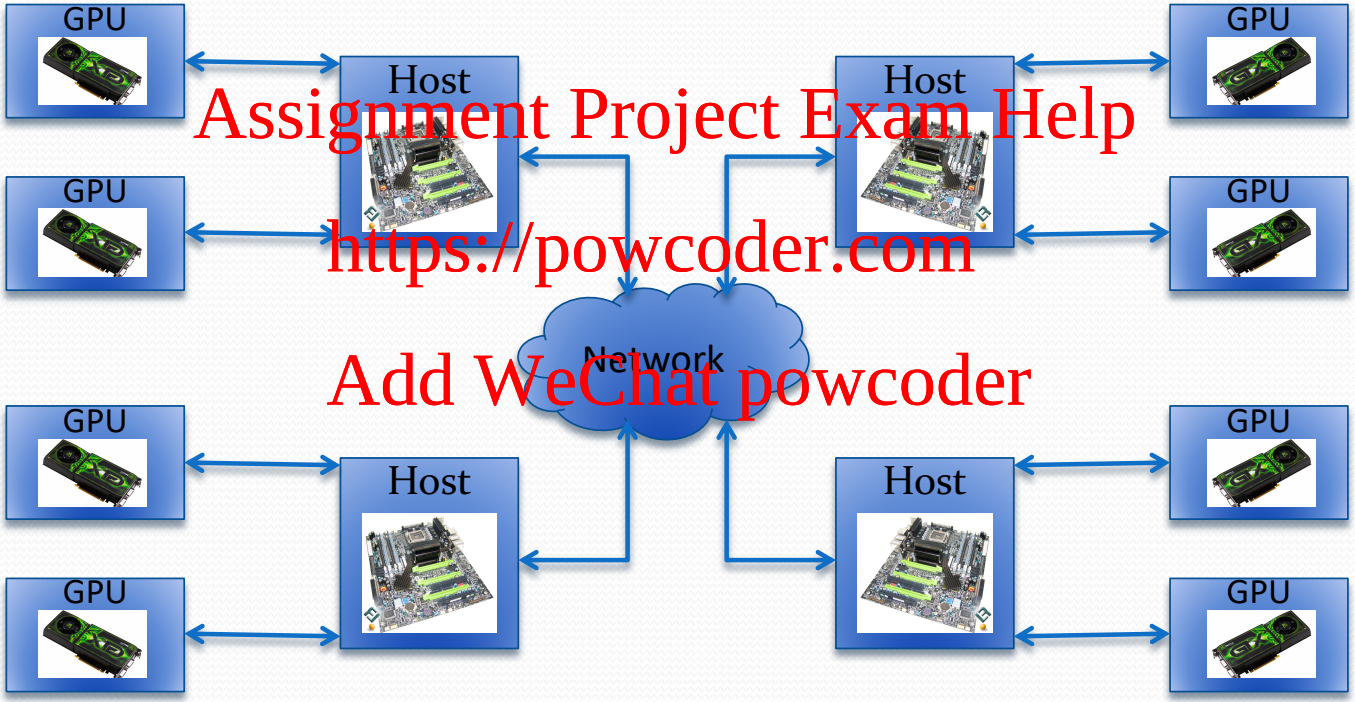
- GPU System Architecture



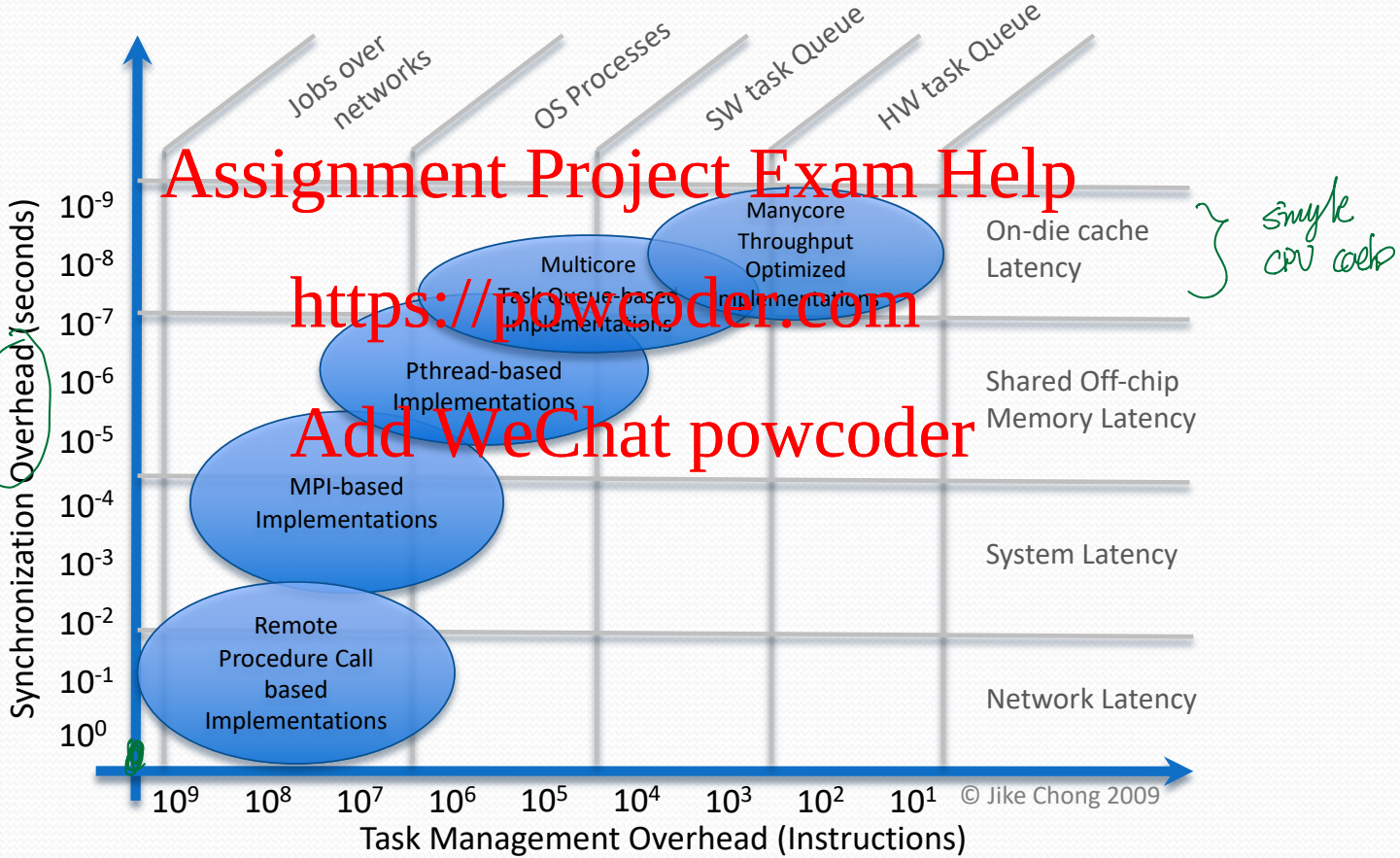
Section 3 – The Cloud



Section 3 – The Cloud



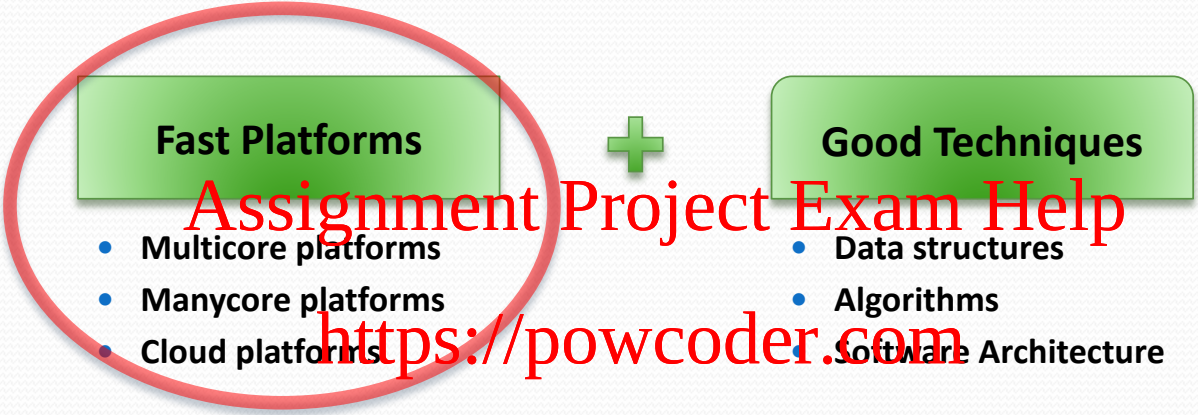
A Game of Granularity



Outline

- Landscape of Computing Platforms
- Hardware Architectures
  - Multicore vs Manycore
  - Instruction level parallelism
  - SIMD
  - Simultaneous multithreading
  - Memory hierarchy
  - System hierarchy
- **How to Write Fast Code?**

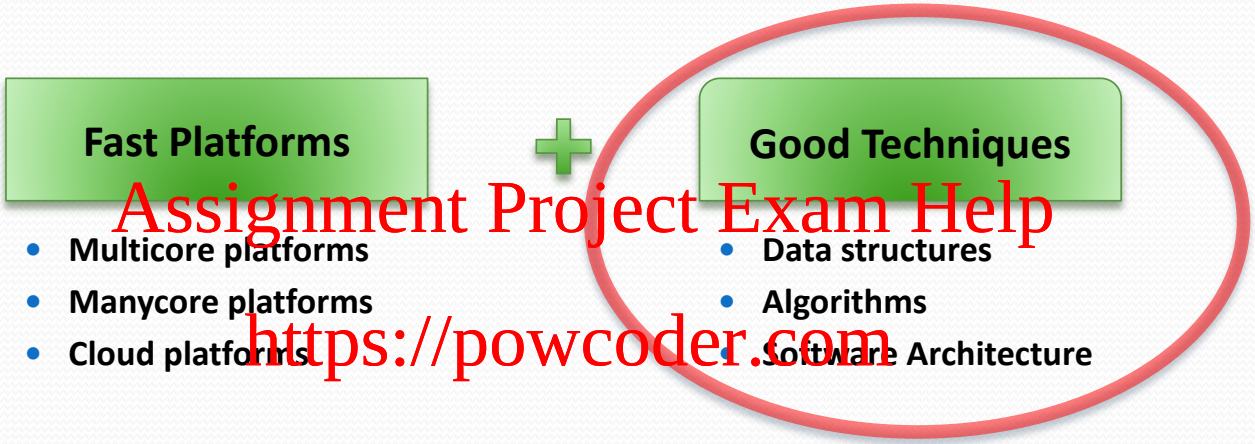
How to Write Fast Code?



- We focus on the fast hardware in this lecture
- Combines with **good techniques** to produce fast code…

  …in order to solve a problem or improve an outcome

How to Write Fast Code?



- Recognizing levels of concurrency in an application
- Effective mapping of concurrency in an application with parallelism of a platform

→ **Fast code**

Concurrency Opportunity Recognition

- **The Application Developer**
- **Application-level Concurrency**
- **The Problem-Solving Process**

Distinction: Concurrency vs. Parallelism

Concurrency	Parallelism
The property of **an application** that…	The property of **a platform** that…
…allows for tasks to **have the potential** to be…	…allows for tasks to **have the potential** to be…
…**executed simultaneously**	…**executed simultaneously**
The **application architecture** in which…	The **platform architecture** in which..
…**more than one task** is active and able to…	…**more than one task** can be active and…
…**make progress** at one time	…**make progress** at same time
We **expose concurrency** in our applications.	We **exploit parallelism** in our platforms.

The Application Developer

- Writing fast code is a process coherent with

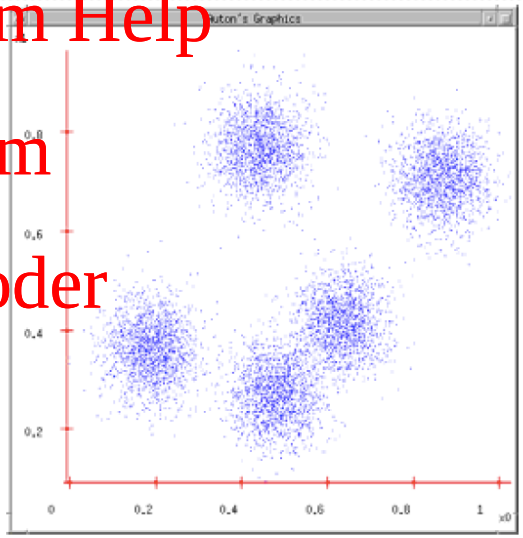
"general problem solving behavior"

Newell and Simon, Human Problem Solving (1972), pp. 72-73

- The process of problem solving involves:
  1. Understand the **current state**
  2. Observe the **internal representation**
  3. **Search** among alternatives
  4. Select from a set of **choices**

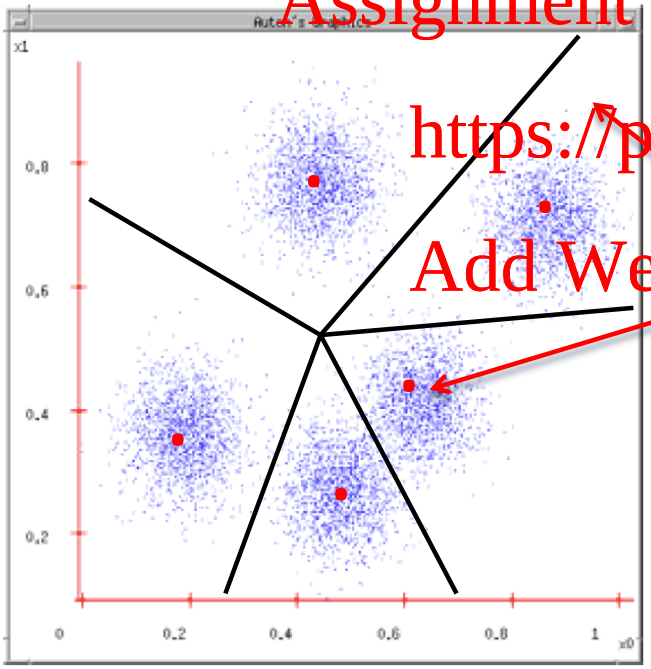
k-means Problem

- Find *k* cluster centers that minimize the distance from each data point to a cluster center
- Important algorithm in machine learning:
  - Statistical data analysis
  - Vector quantization (Speech Recognition)
- NP-hard for arbitrary input
- ***k*-means algorithm** frequently finds a reasonable solutions quickly
- Issues:
  - Worst case running time is super-polynomial
  - Approximation can be arbitrarily bad



k-means Problem

- Find *k* cluster centers that minimize the distance from each data point to a cluster center



Cluster

Cluster center (centroid)

k: Number of clusters (defined a-priori)

Cluster: Assignment of data points to a class

Cluster Center: μ of data points in a cluster

Related Problems and Algorithms

- ***k*-means++**: Maximize scattering on initial cluster centers
- **KD-trees**: Fast *k*-means - Pre-compute distance between data points
- ***x*-means**: *k*-means with efficient estimation of the number of classes

- Gaussian Mixture Models:
  - Probabilistic assignments to clusters
  - Multivariate Gaussian distributions instead of means

- Expectation Maximization algorithms (EM algorithms)
  - Find maximum likelihood estimates of parameters in a statistical model, where the model depends on unobserved latent variables.
- Expectation Maximization Algorithms for Conditional Likelihoods
  - Estimate parameters in a statistical model to optimize conditional likelihood (where the objective function is a rational function)

k-means Algorithm (“*Lloyd’s algorithm*”)

- Given an initial set of *k* means $\mathbf{m}_1^{(1)},\ldots,\mathbf{m}_k^{(1)}$
- **Expectation Step:** Assign each observation to the cluster with the closest mean

$$S_i^{(t)} = \left\{ \mathbf{x}_j : \|\mathbf{x}_j - \mathbf{m}_i^{(t)}\| \leq \|\mathbf{x}_j - \mathbf{m}_{i^*}^{(t)}\| \text{ for all } i^* = 1,\ldots,k \right\}$$

- **Maximization Step:** Calculate the new means to be the centroid of the observations in the cluster.

$$\mathbf{m}_i^{(t+1)} = \frac{1}{|S_i^{(t)}|} \sum_{\mathbf{x}_j \in S_i^{(t)}} \mathbf{x}_j$$

- Iterate until convergence or stopping criteria met

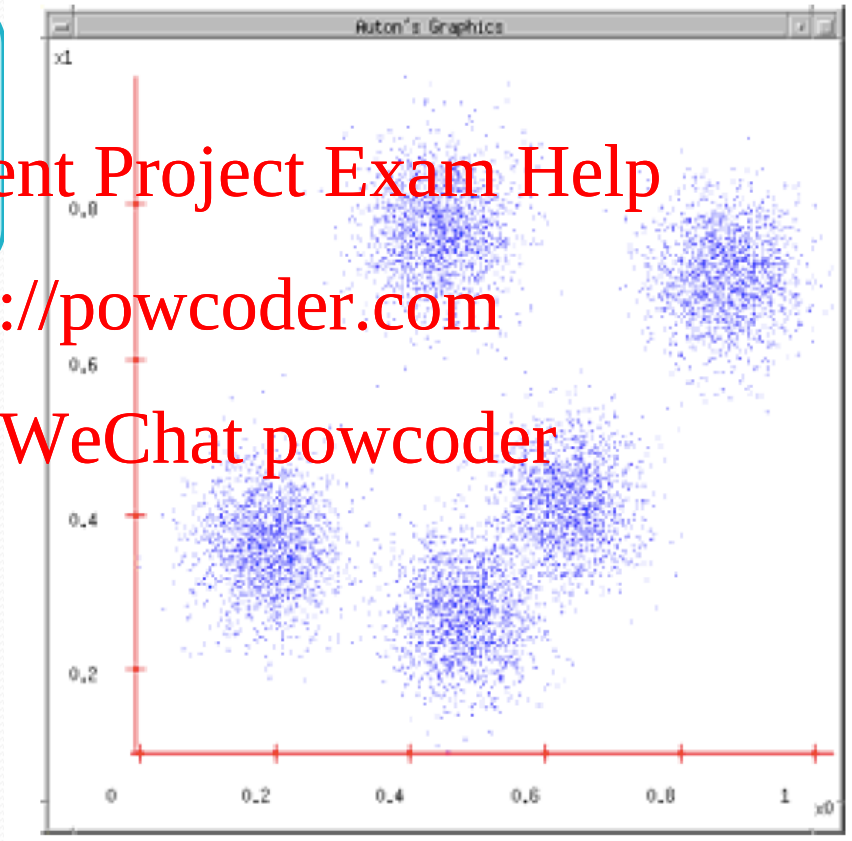
The Algorithm

Example:

k=5

Distance metric=euclidean

Dimensions=2



1. Randomly select k cluster Centers
2. Assign closest Center to each data point
3. Update Centers based on assignments from (2)
4. Re-iterate steps 2-3 until convergence or stopping criteria met

The Algorithm

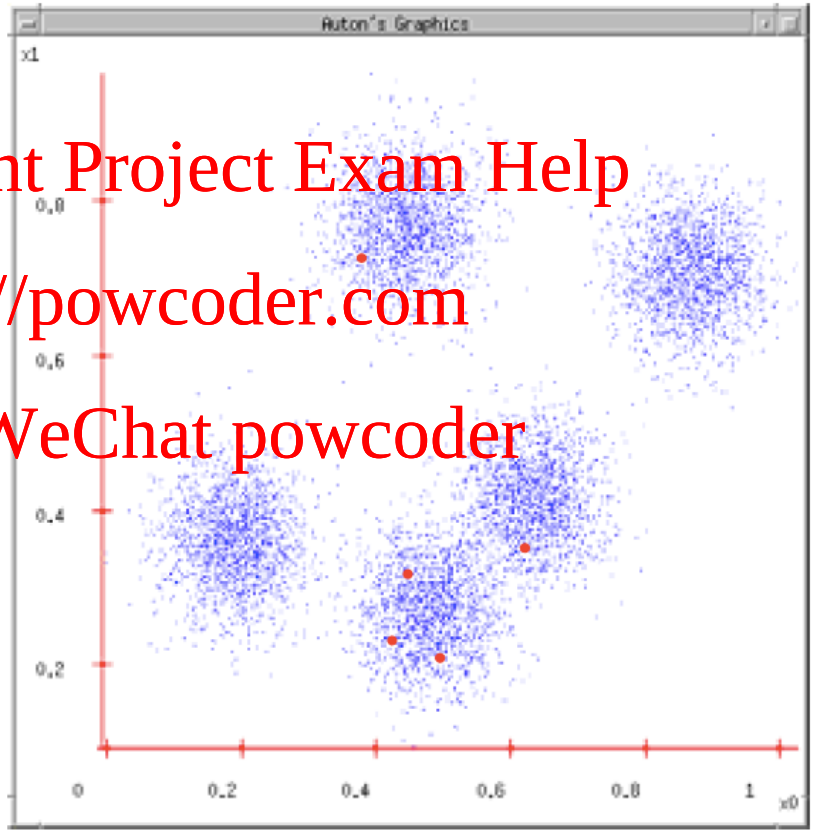
Example:

k=5

Distance metric=euclidean

Dimensions=2

1. Randomly select k cluster Centers
2. Assign closest Center to each data point
3. Update Centers based on assignments from (2)
4. Re-iterate steps 2-3 until convergence or stopping criteria met



The Algorithm

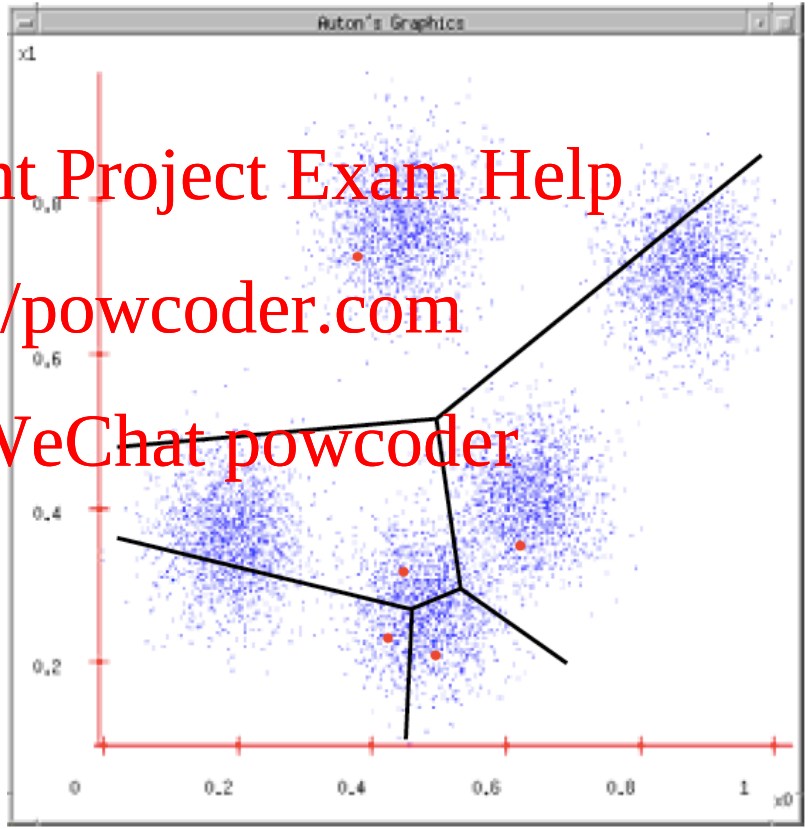
Example:

k=5

Distance metric=euclidean

Dimensions=2

1. Randomly select k cluster Centers
2. Assign each data point to closest Center
3. Update Centers based on assignments from (2)
4. Re-iterate steps 2-3 until convergence or stopping criteria met



The Algorithm

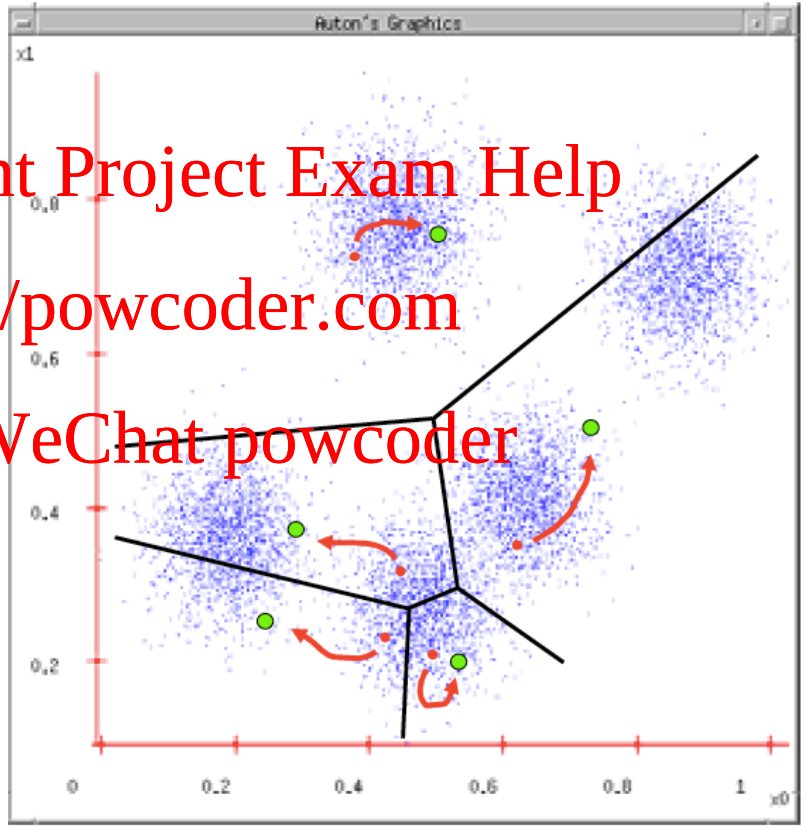
Example:

k=5

Distance metric=euclidean

Dimensions=2

1. Randomly select k cluster Centers
2. Assign closest Center to each data point
3. Update Centers based on assignments from (2)
4. Re-iterate steps 2-3 until convergence or stopping criteria met



The Algorithm

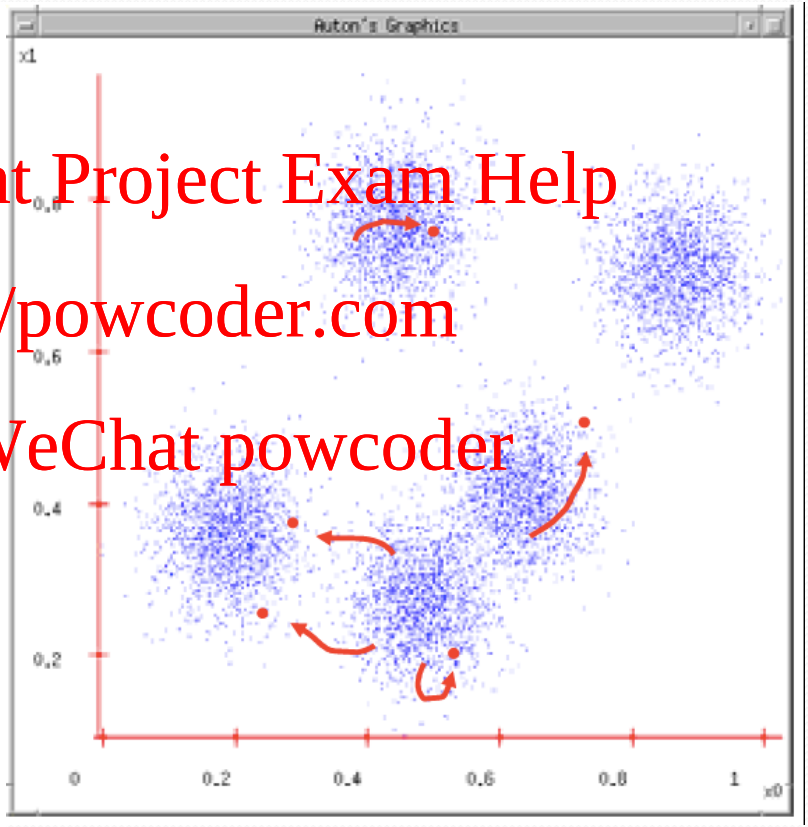
Example:

k=5

Distance metric=euclidean

Dimensions=2

1. Randomly select k cluster Centers
2. Assign closest Center to each data point
3. Update Centers based on assignments from (2)
4. Re-iterate steps 2-3 until convergence or stopping criteria met



The Phases

1. **Initialization:** Randomly select *k* cluster centers
   - Select *k* samples from data as initial centers [Forgy Partition]
2. **Expectation:** Assign each data point go closest center
   - Compare each data point (*N*) to each cluster center (*k*)
   - Distance Metric: Euclidean distance (*D* dimensions)
3. **Maximization:** Update centers based on assignments
   - For each cluster (*k*) compute mean (D dimensions) from data points assigned to that cluster
4. **Evaluate:** Re-iterate steps 2-3 until convergence or stopping criteria met
   - Percentage of data points re-assigned
   - Number of iterations (2-3)

A Fast Implementation of *k*-means

- Following the process of problem solving with *k*-means:
  1. Understand the **current state**
     - Running on a platform
     - Using a specific set of resources
     - Achieving a specific performance
     - Meeting a specific criteria/requirement
  2. Observe the **internal representation**
  3. **Search** among alternatives
  4. Select from a set of **choices**

Assumption:
Starting from a functionally correct reference implementation

Implication:
Must observe the *current state* and *implementation requirements* before starting to solve a problem

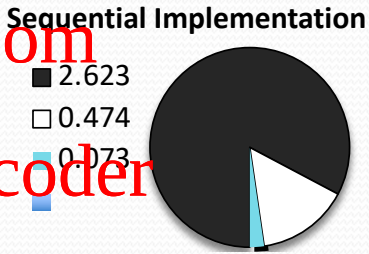
Understanding the Current State

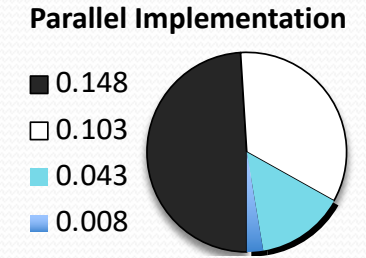
- **Running on a platform**
  - *Platform:* *Linux + GCC on x86 multicore processor*
- **Using a specific set of resources (given X)**
  - *Computation:* *? cores, 2 to ? way SIMD*
  - *Data:* *??KB L1, ??KB L2, shared L3 cache, ??GB DRAM*
  - *Synchronization:* *on-chip shared-memory abstraction*
- **Achieving a specific performance**
  - *As measured in Mini-Project 1*
- **Meeting a specific criteria/requirement**
  - *Matrix-Multiply*
  - *k-means*

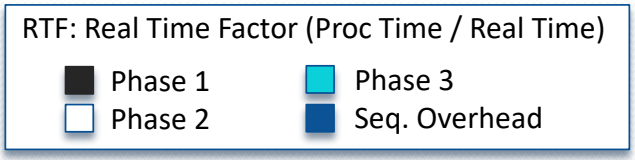
What to Measure for Performance?

- **Performance Analysis: Roofline Model**

- A few simple techniques:
  - Observe the phases of execution
  - Characterize the execution time breakdowns
  - Reason about why a piece of code is slow
  - Identify performance bottlenecks







Current state: *k*-means algorithm

- 4 Phases (Initialization, Expectation, Maximization, Evaluate)
  - Majority of time spent on Expectation and Maximization phases
- Entire data set can fit in memory on a single machine
- Number of samples (*N*) and feature dimensions (*D*) vary significantly
- Evaluation for any number of clusters ($2 \geq k \leq 300$)
- Example Data Sets:
  - *ionosphere_scale*: 351 Samples, 34 Dimensions
  - *svmguide*: 7089 Samples, 4 Dimensions
  - *cod-rna*: 59535 Samples, 8 Dimensions
  - *ljcnn1*: 191681 Samples, 22 Dimensions

A Fast Implementation of *k*-means

- Following the process of problem solving with *k*-means:
  1. Understand the **current state**
  2. Observe the **internal representation**
     - Application structure
       - Identified four phases of execution
     - Implementation concerns
       - Task considerations
       - Data representations
       - Concurrency opportunities
  3. **Search** among alternatives
  4. Select from a set of **choices**

Example Code (Initialize)

kmeans/seq_kmeans.c
lines: 116-119

```
....
    /* pick first numClusters elements of objects[] as initial cluster centers*/
    for (i=0; i<numClusters; i++)
        for (j=0; j<numCoords; j++)
            clusters[i][j] = objects[i][j];
....
```



```
__inline static float euclid_dist_2(int numdims, float *coord1, float *coord2)
{
  int i;
  float ans=0.0;
  for (i=0; i<numdims; i++)
    ans += (coord1[i]-coord2[i]) * (coord1[i]-coord2[i]);
  return(ans);
}
```

Define distance metric (Euclidian)

kmeans/seq_kmeans.c
lines: 51-63

Example Code (Initialize)

kmeans/seq_kmeans.c
lines: 116-119

....

Active Data Structures

objects: N x D

clusters: k x D

...

__inline static float euclid_dist_2(int numdims, float *coord1, float *coord2)
{

Active Data Structures

coord1: 1 x D

coord2: 1 x D

ans: 1

}

Define distance metric (Euclidian)

kmeans/seq_kmeans.c
lines: 51-63

Possible concurrencies:
D (sum reduction) ← euclid_dist_2()

Example Code (Expectation)

kmeans/seq_kmeans.c
lines: 136-147

```
....
    delta = 0.0;
    for (i=0; i<numObjs; i++) {
      /* find the array index of nestest cluster center */
      index = find_nearest_cluster(numClusters, numCoords, objects[i], clusters);

          ...

      /* if membership changes, increase delta by 1 */
      if (membership[i] != index) delta += 1.0;

      /* assign the membership to object i */
      membership[i] = index;
...
```

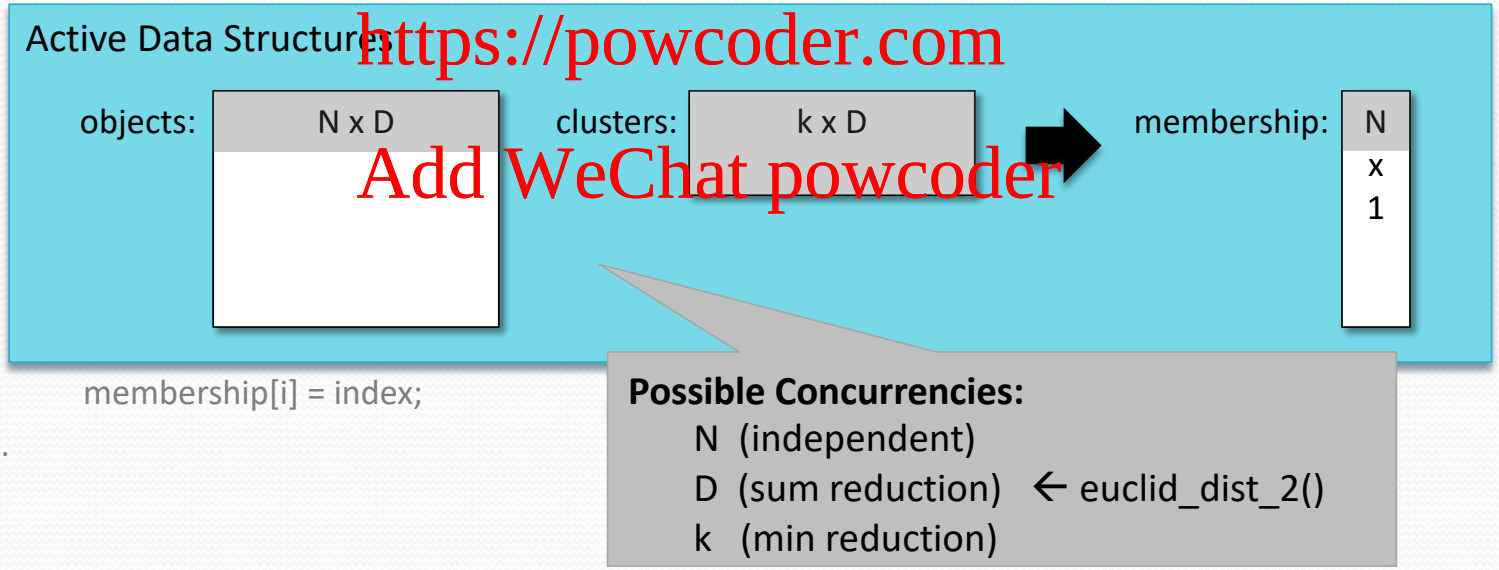
Evaluate distance to each cluster centroid and select closest

Example Code (Expectation)

kmeans/seq_kmeans.c
lines: 136-147

```
....
    delta = 0.0;
    for (i=0; i<numObjs; i++) {
      /* find the array index of nestest cluster center */
      index = find_nearest_cluster(numClusters, numCoords, objects[i], clusters);

      membership[i] = index;
...
```

Example Code (Maximization)

kmeans/seq_kmeans.c
lines: 148-162

```
....
    /* update new cluster centers : sum of objects located within */
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
            newClusters[index][j] += objects[i][j];
    }

    /* average the sum and replace old cluster centers with newClusters */
    for (i=0; i<numClusters; i++) {
        for (j=0; j<numCoords; j++) {
            if (newClusterSize[i] > 0)
                clusters[i][j] = newClusters[i][j] / newClusterSize[i];
            newClusters[i][j] = 0.0;   /* set back to 0 */
        }
        newClusterSize[i] = 0;       /* set back to 0 */
    }
...
```

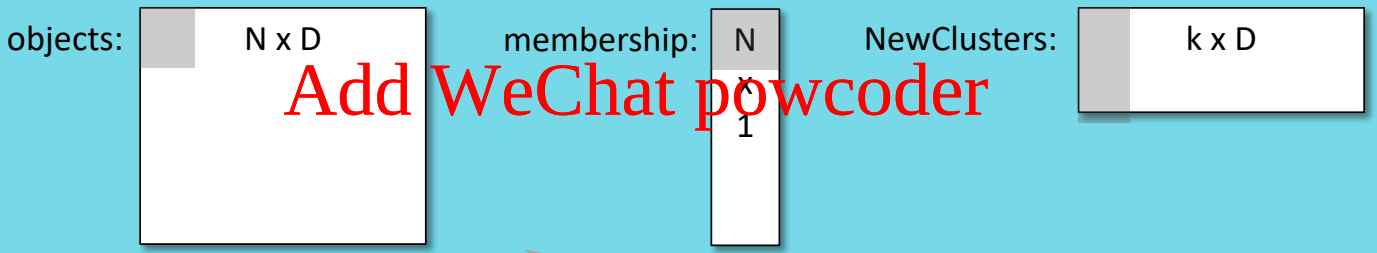
prepare for next iteration

Example Code (Maximization)

kmeans/seq_kmeans.c
lines: 148-162

```
....
    /* update new cluster centers : sum of objects located within */
        newClusterSize[index]++;
        for (j=0; j<numCoords; j++)
          newClusters[index][j] += objects[i][j];
        }
        newClusterSize[i] = 0;      /* set bac
    }
...
```


Active Data Structures



Possible Concurrencies:
D (independent)
N (Histogram computation into k bins

The phases - concurrency

1. **Initialization:** Randomly select *k* cluster centers
2. **Expectation:** Assign closest center to each data point
   - *N* (independent)
   - *k* (min reduction)
   - *D* (sum reduction)
3. **Maximization:** Update centers based on assignments
   - *D* (independent)
   - *N* (Histogram computation into k bins)
4. **Evaluate:** Re-iterate steps 2-3 until convergence

A Fast Implementation of *k*-means

- Following the process of problem solving with *k*-means:
  1. Understand the **current state**
  2. Observe the **internal representation**
  3. **Search** among alternatives
  4. Select from a set of **choices**

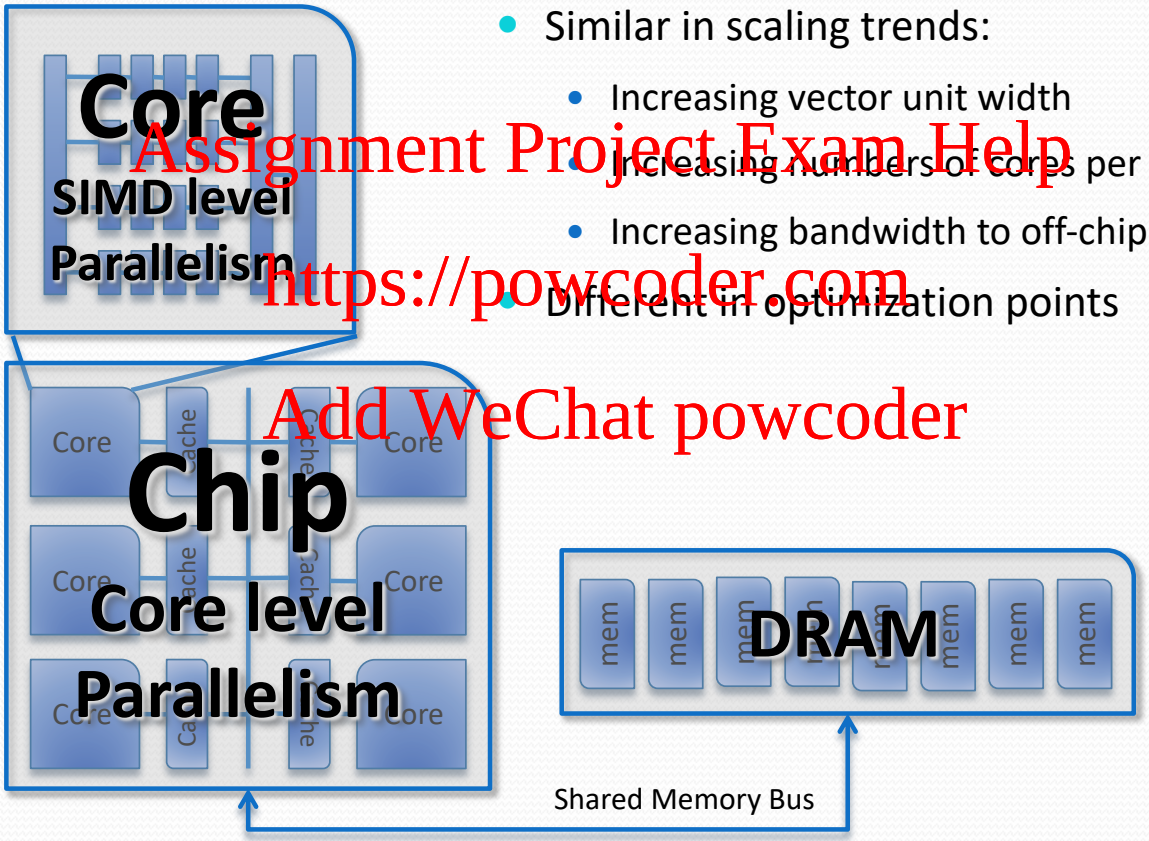
Search Among Alternatives

- Given the observed internal representations and the concurrency opportunities…

- What are the implementation **alternatives**?
  - Different mapping of **application concurrency** to **platform parallelism**

- The **search** process
  - More complex than one **application concurrency** to one **platform parallelism**
  - May want to sequentialize some operations:
    - Some parallel operations are as **"work-efficient"** as sequential operations
    - Reduction – sequential: O(N), Parallel: O(N logN)
  - One level of concurrency could map to multiple levels of parallelism

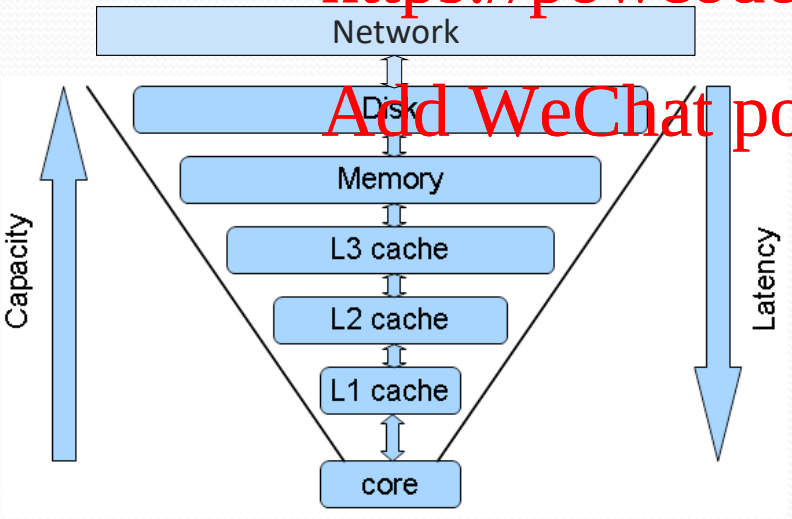
Multicore and Manycore Parallelism



- Similar in scaling trends:
  - Increasing vector unit width
  - Increasing numbers of cores per die
  - Increasing bandwidth to off-chip memory
- Different in optimization points

Memory Hierarchy

- **Cache**:
  - A piece of fast memory close to the compute modules in a microprocessor
  - Allows faster access to a limited amount of data
  - Physical properties of wires and transistors determines trade-off between **cache capacity** and **access throughput/latency**



(Capacity), (throughput), (latency)

Exa-Bytes, 0.1 GB/s, ~1-10B cycles

3GB, 6 GB/s, ~1M cycles

16GB, 34.08 GB/s, ~200 cycles

8 MB, 108.8 GB/s, 26-31 cycles

256 KB, 108.8 GB/s, 12 cycles

32 KB Data Cache, 163.2 GB/s, 4-6 cycles

Intel Core2 – 2600k @ 3.4GHz
(viewed from one core)

Mapping Concurrency to Parallelism

- How does it map to the platform?
  - SIMD level parallelism
  - Core level parallelism
- How does it map to the cache hierarch?
  - What data is required for each concurrent operation?
  - What are the synchronization points in the algorithm?
- Expectation & Maximization Phases
  - SIMD & core-level parallelism across data-points (N)
    - Update membership for each data point sequentially
      - Compute distance to each cluster center and select index with min. distance
    - Histogram computation (summation / assignment count for new clusters)
  - Other possible concurrency mappings?

A Fast Implementation of *k*-means

- Following the process of problem solving with *k*-means:
  1. Understand the **current state**
  2. Observe the **internal representation**
  3. **Search** among alternatives
  4. Select from a set of **choices**
     - Does solution met required criteria
     - How to evaluate a mapping?
       - Efficiency: Runs quickly, makes good use of computational resources
       - Simplicity: Easy to understand code is easier to develop, debug, verify and modify
       - Portability: Should run on widest range of parallel computers
       - Scalability: Should be effective on a wide range of processing elements
     - Other considerations: Practicality, Hardware, Engineering cost

Evaluate Choice

- Expectation & Maximization Phases
  - SIMD & core-level parallelism across data-points (N)
    - Update membership for each data point sequentially
    - Histogram computation (summation / assignment count for new clusters)

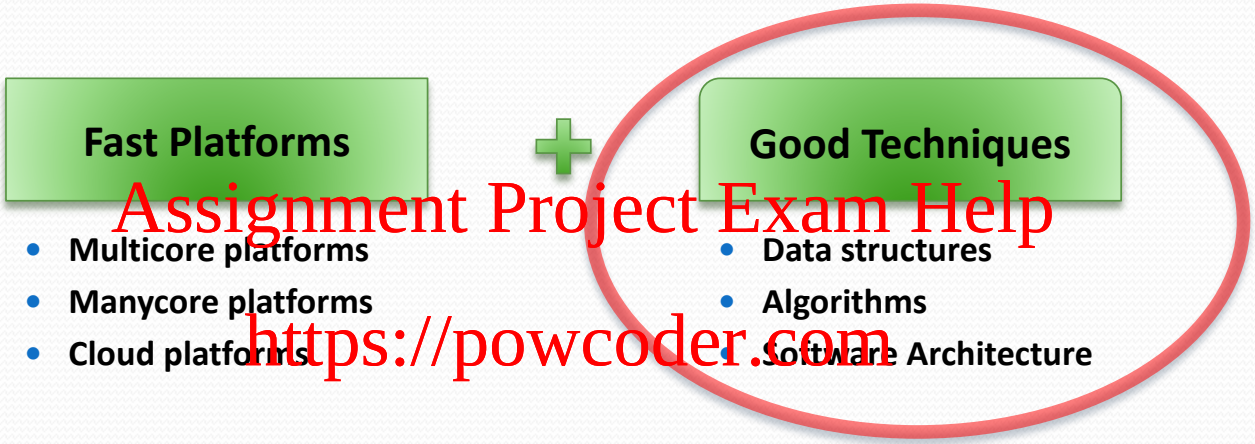
  → OpenMP

- How we can evaluate the choice and make a decision
  - Efficiency
  - Simplicity / Maintainability
  - Portability
  - Scalability

How to write fast code

- **Expose** concurrencies in applications and algorithms
  - 2/9 - "Concurrency Opportunity Recognition"
  - Mini-Projects (1-3) & Term Project
- **Exploit** parallelisms on application platform
  - 2/4 - "Advanced Parallel Hardware Architectures"
  - Mini-Projects (1-3) & Term Project
- **Explore** mapping between concurrency and parallelism
  - The rest of the semester….
  - Abstractions to support mapping of concurrencies to parallelisms
    - OpenMP [**Section 1**]
    - CUDA [**Section 2**]
    - Map-Reduce [**Section 3**]

How to Write Fast Code?



- Recognizing levels of concurrency in an application
- Effective mapping of concurrency in an application with parallelism of a platform

→ **Fast code**