Week 5: Concurrent Designs & Patterns

MPCS 52060: Parallel Programming

University of Chicago

Parallel Designs & Patterns

Designing Parallel Programs: Initial Steps[1]

The first step to designing parallel programs is to **understand the problem you are trying to solve**.

Identify whether the problem can be parallelized:

- For example – "Calculate the potential energy for each of the several thousand independent conformations of a molecule. When done, find the minimum energy conformation."

  Yes each of the molecular conformations is independently determinable. The calculation of the minimum energy conformation can be done with a parallel reduction.

[1] Blaise Barney, Lawrence Livermore National Laboratory

Designing Parallel Programs: Initial Steps[2]

- For example – Calculation of the Fibonacci series (0,1,1,2,3,5,8,13,21,...) by use of the formula:

  $$F(n) = F(n-1) + F(n-2)$$

  - **No**, the calculation of the F(n) value uses those of both F(n-1) and F(n-2), which must be computed beforehand. Very little (to no way) to parallelize this problem.

Additionally, if you are starting with a serial program, this means you need to ensure you understand that sequential code.

[2] Blaise Barney, Lawrence Livermore National Laboratory

Hotspots & Bottlenecks[3]

When designing a parallel program, its important to be aware of a program's hotspots and bottlenecks:

- Hotspots – the areas within your program that are doing the most work.
  - In most scientific/technical applications, there are only a few hotspots.
  - Profilers and performance analysis tools can help identify these areas.
  - Focus on parallelizing the hotspots and ignore those sections of the program that account for little CPU usage.

[3] Blaise Barney, Lawrence Livermore National Laboratory

Hotspots & Bottlenecks[4]

- Bottlenecks – are areas of a program that are causing a slow down in performance.
  - For example: I/O system call usually will cause a slow down in performance.
  - Think about using different algorithms to help remove unnecessary slow areas.
- Identify inhibitors to parallelism.
  - For example, data dependence between sections of code such as with the Fibonacci sequence example.

[4] Blaise Barney, Lawrence Livermore National Laboratory

Partitioning[5]

Partitioning (also known as decomposition) is breaking down a problem into discrete "chunks" of work that can be distributed to multiple threads.

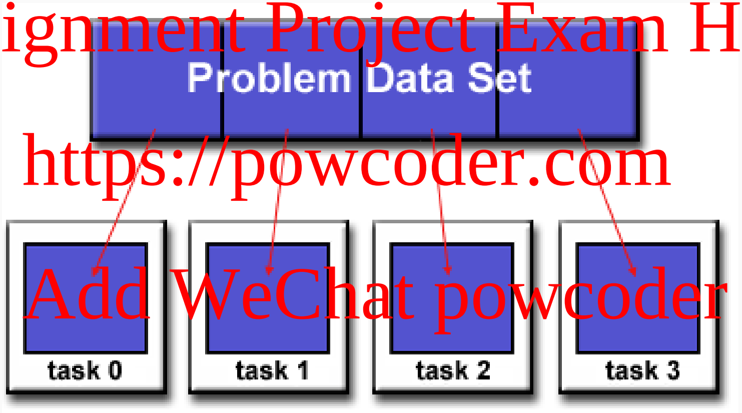
The two main ways to partition computational work among threads:

- Data Decomposition (data parallelism) – the data associated with a problem is decomposed. Each thread then works on a portion of the data
  - Many numerical analysis and scientific computing algorithms are based on vectors and matrices, which are represented by one-, two-, or higher dimensional arrays.
  - Fairly straightforward to decompose array-based data into subarrays and assign the subarrays to different threads

[5] Blaise Barney, Lawrence Livermore National Laboratory

Partitioning[6]

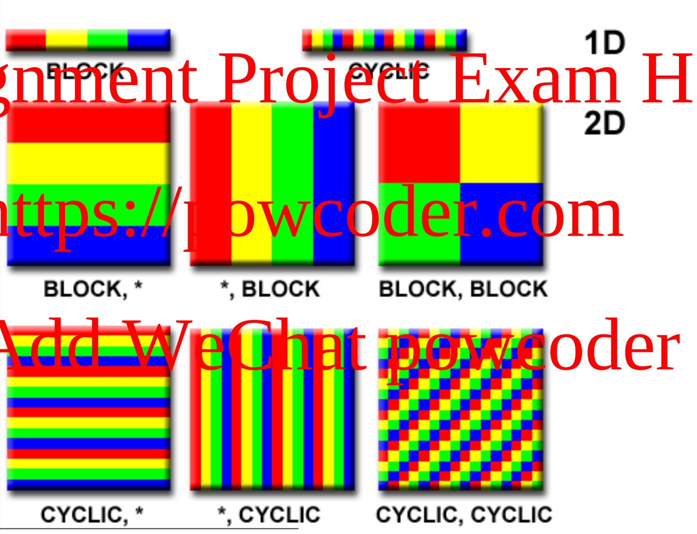
High-level example of Data Decomposition:



[6] Blaise Barney, Lawrence Livermore National Laboratory

Data Decomposition Partitioning[7]

- The distribution of data to threads can be done in various ways:



[7] Blaise Barney, Lawrence Livermore National Laboratory

Data Decomposition Partitioning (cont.)[8]

- Which distribution mapping to choose based on efficient memory access
  - A unit stride (stride of 1) maximizes cache/memory usage.
  - Algorithms may access elements in a specific way.
- The choice of a distribution mapping depends on the programming language and how it performs unit striding.

[8] Blaise Barney, Lawrence Livermore National Laboratory

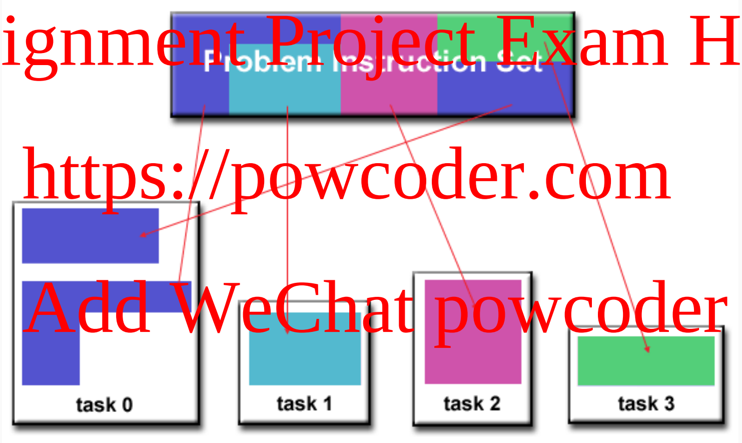
Partitioning[9]

- Functional Decomposition (task parallelism) – the focus is on a program's computation instead of the data that is manipulated by the computation.
    - Many sequential programs contain code sections that are independent of each other, where these code sections can be composed of single statements, basic blocks, loops, or function calls.
    - Independent code sections are known as tasks that are distributed to threads. Thus, each thread performs a portion of the overall work.
    - Example: PI Calculation from homework #2 and Task Queue from Project #1

[9] Blaise Barney, Lawrence Livermore National Laboratory

Partitioning[10]

High-level example of Functional Decomposition:



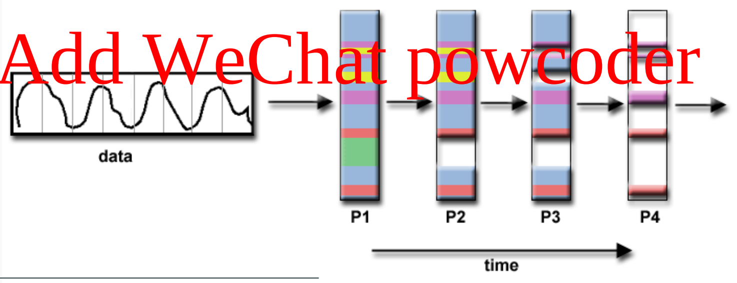
[10] Blaise Barney, Lawrence Livermore National Laboratory

Functional Decomposition Partitioning[11]

Functional decomposition works the best for problems that can be split into different tasks:

- Signal Processing

  "An audio signal data set is passed through four distinct computational filters. Each filter is a separate [thread/process]. The first segment of data must pass through the first filter before progressing to the second. When it does, the second segment of data passes through the first filter. By the time the fourth segment of data is in the first filter, all four tasks are busy."



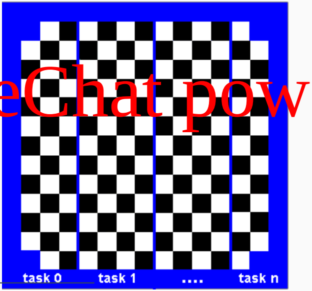
[11] Blaise Barney, Lawrence Livermore National Laboratory

Communication[12]

The point at which synchronization needs to happen between threads/processes executing tasks is dependent on the type of problem being solved:

- Embarrassingly Parallel Problems
    - These types of problems require virtually no need for threads to share data between each other (i.e., little-to no synchronization is required).

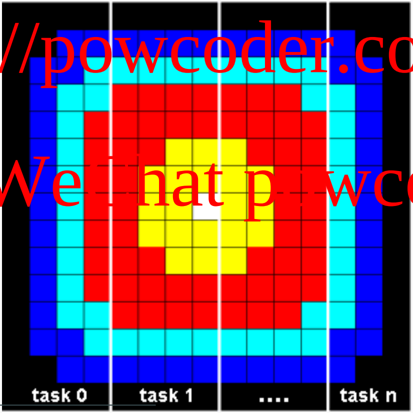
    - Example: image processing, rendering pixels in video games,



[12] Blaise Barney, Lawrence Livermore National Laboratory

Communication[13]

- Data-Dependent Parallel Problems
  - These types of problems require tasks to share data with each other
  - Example: 2-D heat diffusion - requires a task to know the temperatures calculated by the tasks that have neighboring data. Changes to neighboring data has a direct effect on that task's data.



[13] Blaise Barney, Lawrence Livermore National Laboratory

Communication Factors[14]

Consider the following factors when communication with other threads:

- Communication overhead
    - “Inter-task communication virtually always implies overhead.”
    - “Machine cycles and resources that could be used for computation are instead used to package and transmit data.”
    - “Communications frequently require some type of synchronization between tasks, which can result in tasks spending time ”waiting” instead of doing work.”
    - “Competing communication traffic can saturate the available network bandwidth, further aggravating performance problems.”

[14] Blaise Barney, Lawrence Livermore National Laboratory

Communication Factors (cont.) [15]

- Synchronous vs. Asynchronous Communications
  - Synchronous communications require an acknowledgement between both threads before proceeding (known as *blocking*)
  - Asynchronous Communications allow threads to transfer data independently from one another (Example: Project 1 (Part 3)).
  - *Non-blocking* communication is also known as asynchronous communication.
  - **“Interleaving computation with communication is the single greatest benefit for using asynchronous communications”.**

[15] Blaise Barney, Lawrence Livermore National Laboratory

Data Dependencies

Terminology[16]:

- 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 important to parallel programming because they are one of the primary inhibitors to parallelism.

[16] Blaise Barney, Lawrence Livermore National Laboratory

Data Dependency Example

Loop carried data dependence

```go
var data []int
data = ... // fill data
for i := 1; i < len(data); i++ {
    data[i] = data[i-1] * 2
}
```

Parallelism is inhibited because the value of *data(i-1)* must be computed before the value of *data(i)*, therefore *data(i)* exhibits a data dependency on *data(i-1)*.

Data Dependency Example

Loop carried data independence

```
var data []int
data = ... // fill data
for i := 1; i < len(data); i++ {
    data[i] = data[i] * fib(22)
    print(longComputingTask(data[i]))
}
```

Parallelism is not inhibited. There exist no data dependencies within this code so potentially this could be parallelized.

Data Dependency Example

Loop independent data dependence

```
Thread 1      Thread 2
------      ------
X = 9       X = 5
  .           .
  .           .
Y = X + 2   Y = X * 3
```

Parallelism is inhibited. The value of *Y* is dependent on which thread last stores the value of *X*.

Why are Data Dependencies Important?

“Although all data dependencies are important to identify when designing parallel programs, loop carried dependencies are particularly important since loops are possibly the most common target of parallelization efforts”[17].

You can handle data dependencies by synchronizing read/write operations between threads.

[17] Blaise Barney, Lawrence Livermore National Laboratory

Load Balancing

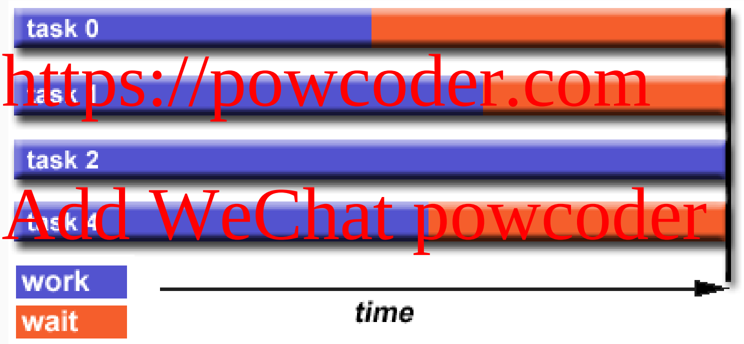
Terminology:

- Load balancing refers to the practice of distributing approximately equal amounts of work among tasks so that **all** tasks are kept busy all of the time. It can be considered a minimization of task idle time.[18]
- Good load balancing with a parallel program leads to better performance.

[18] Blaise Barney, Lawrence Livermore National Laboratory

Load Balancing

- For example, the main thread waiting on a waitgroup of threads to complete. The slowest thread will determine the overall performance.



Load Balancing

How do you achieve Load Balancing?

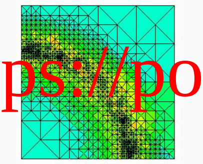
- Equally Partitioning[19]:
  - For array/matrix operations where each task performs similar work, evenly distribute the data set among the tasks.
  - For loop iterations where the work done in each iteration is similar, evenly distribute the iterations across the tasks.
  - If a heterogeneous mix of machines with varying performance characteristics are being used, be sure to use some type of performance analysis tool to detect any load imbalances. Adjust work accordingly.

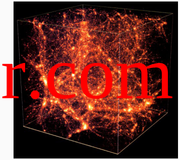
[19] Blaise Barney, Lawrence Livermore National Laboratory

Load Balancing

How do you achieve Load Balancing?

- Dynamic Partitioning[20]:
  - Certain classes of problems result in load imbalances even if data is evenly distributed among tasks





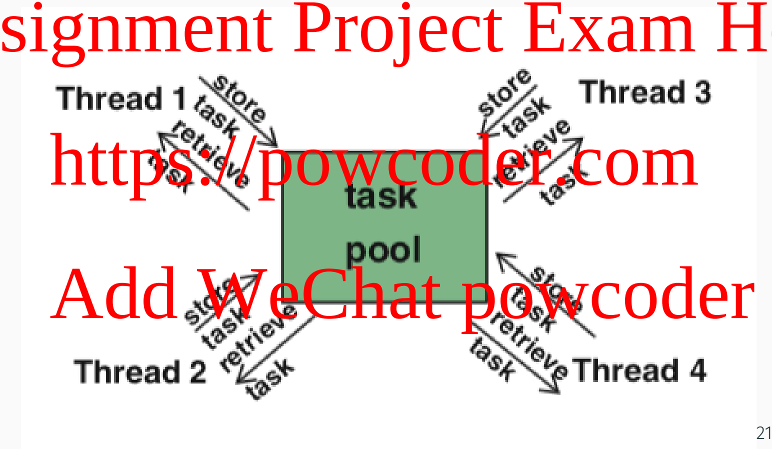
Adaptive Grid Methods (provide numerical solutions of partial differential equation (PDE) with high accuracy): some tasks may need to refine their mesh while others don't.

N-body simulations: particles may migrate across task domains requiring more work for some tasks.

[20] Blaise Barney, Lawrence Livermore National Laboratory

Load Balancing

If the amount of work is unpredictable then using a scheduler-task pool(i.e., task queue) approach might be helpful.



[21]

[21] Rauber,Rünger:Parallel Programming 2013

Number of threads & Sequentialization

When designing a parallel program it is important

- To use a suitable number of threads which should be selected according to the degree of parallelism provided by the application and the number of execution resources available
  - Depending on the thread execution model, the number of threads created should not be too large to keep the overhead for thread creation, management and termination small.
  - Performance degradations may result, if too many threads share the same resources because a degradation of the read/write bandwidth might result.
- To avoid sequentialization by synchronization operations whenever possible.
  - Too many synchronizations may lead to only a small number of threads being active because other threads are waiting because of a synchronization operation.

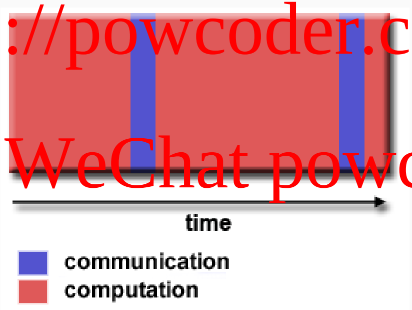
Granularity

The granularity of a parallel program is related to computation time of a task.

- Specifically, it's the qualitative measure of the ratio of computation to communication.
- The granularity of a parallel program is typically considered being:
  - Coarse-grain: tasks with many computations
  - Fine-grain: tasks with only a few computations

Coarse-grain Granularity[22]

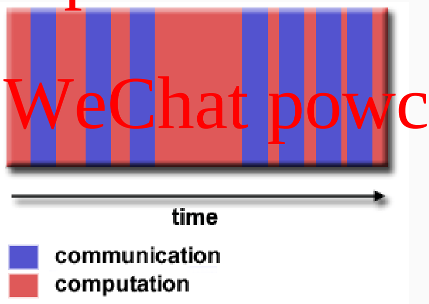
- Relatively large amounts of computational work are done between communication/synchronization events
- High computation to communication ratio
- Implies more opportunity for performance increase
- Harder to load balance efficiently



[22] Blaise Barney, Lawrence Livermore National Laboratory

Fine-grain Granularity[23]

- Low computation to communication ratio
- Facilitates load balancing
- Implies high communication overhead and less opportunity for performance enhancement
- If granularity is too fine it is possible that the overhead required for communications and synchronization between tasks takes longer than the computation.



[23] Blaise Barney, Lawrence Livermore National Laboratory

Coarse-grain vs Fine-grain Granularity[24]

How do you know what one to use? Depends on the algorithm and hardware environment

- If the overhead associated with communications and synchronization is relatively high then coarse-grain might be the better option.
- Fine-grain can help with load imbalance due to a reduction in overhead costs.

[24] Blaise Barney, Lawrence Livermore National Laboratory

Input and Output[25]

- Reduce overall I/O as much as possible (Most important).
- Writing large chunks of data rather than small chunks is usually significantly more efficient.
- Fewer, larger files perform better than many small files.

[25] Blaise Barney, Lawrence Livermore National Laboratory

Input and Output[26]

I/O operations can be a significant parallelism inhibitors. Here are some helpful pointers:

- Confine I/O to specific serial portions of the job, and then use parallel communications to distribute data to parallel tasks.
- Aggregate I/O operations across tasks - rather than having many tasks perform I/O, have a subset of tasks perform it.

[26] Blaise Barney, Lawrence Livermore National Laboratory