CS 6068/5168
Parallel Computing
Module 1

Prof. Fred Annexstein
fred.annexstein@uc.edu
Office Hours: Wednesday 3-4pm on Zoom

Module 1: Welcome

- Goals of this course
- Syllabus, policies, grading
- Canvas Resources
- Introduction/Motivation
- History and Trends in high performance computing
- Scope of the Problems in Parallel Computing

Course Goals

Learning Outcomes:

• Students will learn the computational thinking
and programming skills needed to achieve terascale computing performance, applicable in all science and engineering disciplines.

• Students will learn algorithmic design patterns for parallel computing, critical system and architectural design issues, and programming methods and analysis for parallel computing software.

Workload and Grading Policy

The grading for the class will be based on 2 exams and 6-8 lab projects and homework assignments.

Each student’s final grade will be a weighted

average – Exams 50%, Homework and Labs 50%. Late homework will be accepted with 10 point penalty, and 10 point for each week past due date.

Group work is allowed, but each student submits their own work with references

Course Schedule

- The course calendar is subject to change.
- Part I: focus on high-level functional approaches for multicore computers,
- Part II: focus on CUDA for programming heterogeneous cores,
- Part III: focus on message passing on cluster computers and parallel application areas.
- Variety of Labs will be assigned. You are responsible for programming platform to complete labs.

Course Schedule

Week 1: Introduction to Parallel Computing.

Week 2: Parallel Systems and Architectures.

Week 3: Parallel Algorithmic Models.

Week 4: Parallel Algorithmic Problems.

Week 5: Parallel Programming and Performance.

Week 6: Advanced Memory Models.

Week 7: Parallel Scheduling.

Week 8: Load balancing, Mapping, and Parallel Scalability Analysis.

Week 9: Parallel Program Development.

Week 10: Parallel Applications in Message Passing Systems.

Week 11: Alternative and Future Architectures and Methods.

Weeks 12-14: Advanced Topics

Course Platform and Materials

Textbook:

- Parallel Computing by Dr. Fred Annexstein

Recommended Texts:

- CUDA by Example, Addison-Wesley
- Introduction to Parallel Programming by Peter Pacheco
- Programming on Parallel Machines by Norm Matloff (free online)

Lab Equipment:

- Your own hardware Linux, Mac or PC with a CUDA enabled GPU
- Ohio Supercomputer Center Accounts

Cluster Computing

As a leader in high performance computing and networking, OSC is a vital resource for Ohio's scientists and engineers. OSC's cluster computing capabilities make it a fully scalable center with mid-range machines to match those found at National Science Foundation centers and other national labs.

OSC provides statewide resources to help researchers making discoveries in a vast array of scientific disciplines. Beyond providing shared statewide resources, OSC works to create a user-focused, user-friendly environment for our clients.

Collectively, OSC supercomputers provide a peak computing performance of 7.5 Petaflops (PF). The center also offers approximately 16 Petabytes (PB) of disk storage capacity distributed over several file systems, plus more than 14 PB of available backup tape storage (with the ability to easily expand to over 23 PB).

Technical Specifications

- Pitzer Cluster: A 10,240-core Dell Intel Gold 6148 + 19,104-core Dual Intel Xeon 8268 machine
  - 224 nodes have 40 cores per node and 192 GB of memory per node
  - 340 nodes have 48 cores per node and 192 GB of memory per node
  - 32 nodes have 40 cores, 384 GB of memory, and 2 NVIDIA Volta V100 GPUs
  - 42 nodes have 48 cores, 384 GB of memory, and 2 NVIDIA Volta V100 GPUs
  - 4 nodes have 48 cores, 768 GB of memory, and 4 NVIDIA Volta V100s w/32GB GPU memory and NVLink
  - 4 nodes have 80 cores and 3.0 TB of memory for large Symmetric Multiprocessing (SMP) style jobs
  - Theoretical system peak performance of 3.9 petaflops
- Owens Cluster: A 23,392-core Dell Intel Xeon E5-2680 v4 machine
  - 648 nodes have 28 cores per node and 128 GB of memory per node
  - 16 nodes have 48 cores and 1.5 TB of memory for large Symmetric Multiprocessing (SMP) style jobs
  - 160 nodes have 28 cores, 128 GB of memory, and 1 NVIDIA Pascal P100 GPU
  - Theoretical system peak performance of 1.5 petaflops
- Ascend Cluster: A 2,304-core Dell AMD EPYC™ machine
  - 24 nodes have 88 usable cores, 921GB of usable memory and 4 NVIDIA A100 GPUs per node
  - Theoretical system peak performance of 2.0 petaflops
- GPU Computing: All OSC systems now support GPU Computing. Specific information is given on each cluster's page.
  - Owens: 160 NVIDIA Tesla P100
  - Pitzer: 64 NVIDIA Volta V100 (two each on 32 nodes); 84 NVIDIA Volta V100 (two each on 42 nodes); 16 NVIDIA Volta V100s and NVLink (four each on 4 nodes)
  - Ascend: 96 NVIDIA A100s and and NVLink (four each on 24 nodes)

Shared versus Distributed Memory in Multicore Processors

- Shared memory
  Ex: Intel Core i9 Processors
  - 8-16 Cores
  - One copy of data shared along many core
  - Atomicity, locking and synchronization essential for correctness
  - Many scalability issues
- Distributed memory
  - Ex: Cell Processor
  - Cores primarily access local memory
  - Explicit data exchange between cores
  - Data distribution and communication orchestration is essential for performance

Programming Shared Memory Processors

- Processors 1,2,…,n all ask for X
- There is only one place to look
- Communication through shared variables
- Race conditions possible
- Use synchronization to protect from conflicts
- Change how data is stored to minimize synchronization

Classic Examples of Parallelization

- Data parallelism

  Perform same computation but operate on different data

- Control parallelism
  - A single process can fork multiple concurrent threads
  - Each thread encapsulate its own execution path
  - Each thread has local state and shared resources
  - Threads communicate through shared resources such as global memory

Language based approach to concurrency and parallelism

- **Concurrency** is concerned with managing access to shared state from different threads,
- **Parallelism** is concerned with utilizing multiple processors/cores to improve the performance of a computation.
- Several languages including Python and Clojure have successfully improved the state of concurrent programming with many concurrency primitives and the same for multi-core parallel programming.

Designing Parallel Programs

The initial design of parallel programs is often based on a series of high level operations, which must be carried out for the program to perform the job correctly without erroneous results. These high level operations that must be carried out for parallelization of an algorithm are as follows: Task decomposition, Agglomeration, and Assignment and Mapping.

Task decomposition is where the software program is divided into discrete tasks or a set of instructions that can then be executed on different processors to implement parallelism.

Agglomeration of tasks is the process of combining smaller tasks with larger ones in order to improve performance though increased granularity. Granularity is measured by the ratio of the amount computation to the amount or cost of communication.

Task Assignment and Mapping requires a mechanism to specifiy how subtasks are distributed among the various processors or processes. This phase establishes the distribution of workload among the various processors, and it determines the level of load balancing across the application. Poor load balancing can result in processors existing in an idle state for a long time.

PARALLEL PERFORMANCE METRICS

Designers and developers need performance metrics in order to decide whether designs will be successful. To facilitate performance metric analysis we will compare the parallel algorithm obtained to the original, sequential algorithm from where it is derived.

We now introduce a few performance metric indexes: Speedup, Efficiency, Scaling.

SPEEDUP

Parallel speedup is simply a ratio that measures the benefit of decreased time of solving a problem in parallel. It is defined as the ratio of the time taken to solve a problem sequentially on a single processor (Ts) to the time required to solve the same problem on p identical processors (Tp). We denote speedup as follows:

$$S = Ts / Tp$$

If S=p, then it means that the speed of execution increases with the number of processors and we call it **linear speedup**. This can occur in an ideal case of embarrassingly parallel tasks without communication. A worthy goal is to design parallel programs that have speedup that is proportional to p.

EFFICIENCY

It is usually not reasonable to expect linear speedups, since time is often wasted in either idling or communicating. Efficiency, denoted by Eff, is a measure of how much of the execution time a processing element puts toward doing useful work, given as a fraction of the time spent. Efficiencies are always measured Eff <=1, and algorithms with linear speedup will have an efficiency value of Eff = 1.

In general we have the formula,

$$Eff = S / p = Ts / p\ Tp$$

SCALABILITY

Scalability is a metric defined as the ability to remain efficient on a parallel machine as p increases. By increasing the size of the problem and, at the same time, the number of processors to which we assign tasks, a scalable system will present no loss in terms of speedup performance. A scalable system must maintain the same efficiency or potentially improve it as the size of the problem grows.

For example: Suppose a program can be sped up if parallelized and run on multiple CPUs instead of one. Let a be the fraction of a calculation that is sequential, and 1-a be the fraction that can be parallelized. Then the maximum speedup that can be achieved by using P processors is

$$1 / (a + (1-a)/P)$$

We can apply this formula to check whether adding more processors can make a substantial difference in speedup and thus be considered scalable.

Execution Time and Computational Work

- Computational work = load, Speed is work per unit time
- We assume constant computational speed measured as work per time unit
    - 100 units that runs 50 sec with core rated at 2 units per seconds
    - Therefore speed = 2 units work per second
- Execution time = Work / Speed
- Rendering images: Work = # frames, Speed = #frames/sec
- Instructions per sec has the problem interpretations
- No Universal measure of work
    - MIPS - variety of instruction set families - poor for comparisons
    - Floating Point Operations = FLOP
    - Double precision FLOP per second is standard measure for core execution speed

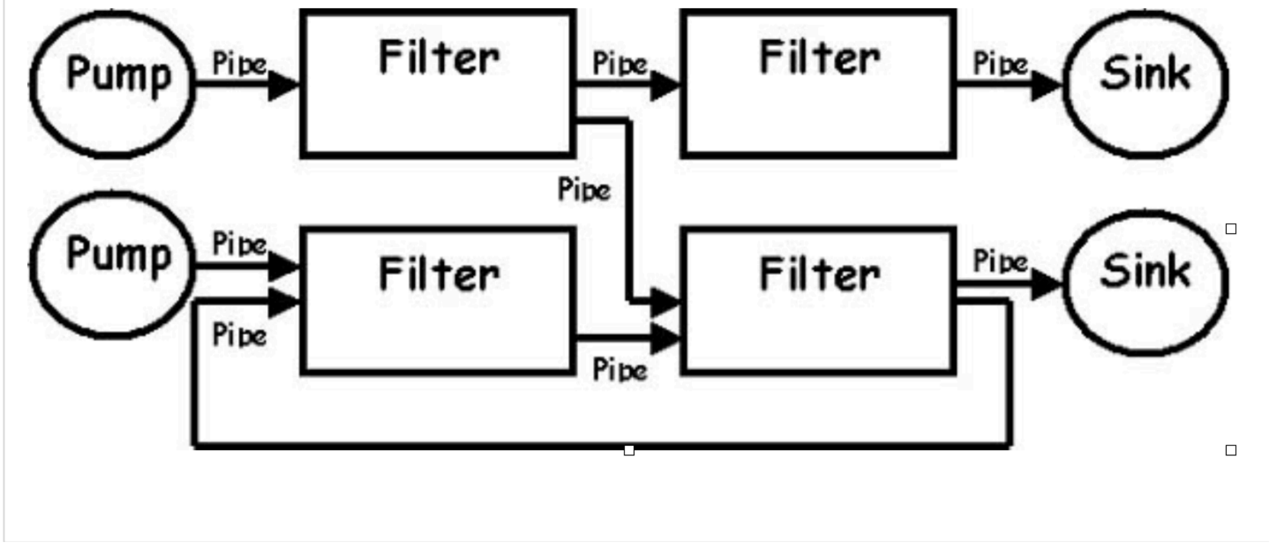
Predicting Execution Time

Example Time = Work / Speed

- Program requires work of 100 TeraFLOP
    - with core executing at rate 35 GigaFLOP per second
- Thus would have 2.8 thousand second execution time
- (100*1e12) / (35*1e9) = 2857.1428571428573 seconds
- Timesharing/Multiprogramming on single core has overhead
- Example of Two programs on single 1 GigaFLOP
- First program: 5-units work; Second program: 10-units work
- Result in timesharing: First finishes at time 10 and Second finishes at time 15

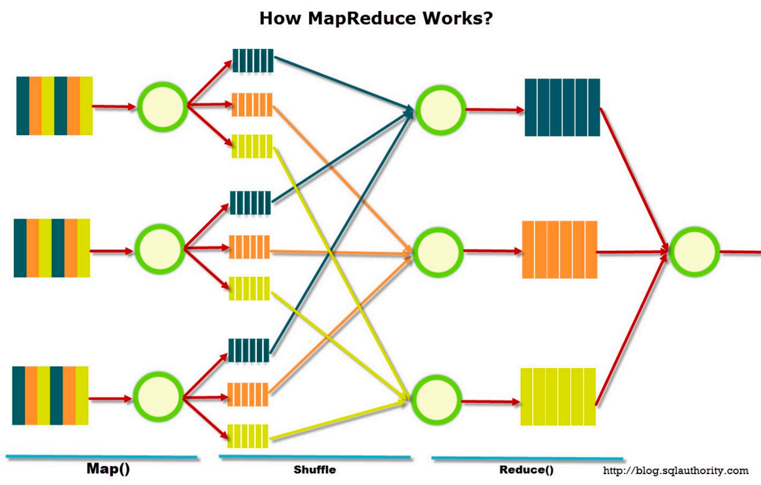
Four Structural Patterns In Parallel Software

Pipe-and-filter: this pattern is used to solve problems characterized by data flowing through modular phases of a computation. The solution constructs the program as filters (computational elements) connected by pipes (data communication channels).



Map-Reduce: the map-reduce pattern is used to solve problems where the same function is independently applied across distributed data. The main issue here is to optimally exploit the computational efficiency and parallelism latent in this structure. The solution is to define a program structured as two or three distinct phases: map, shuffle, reduce.

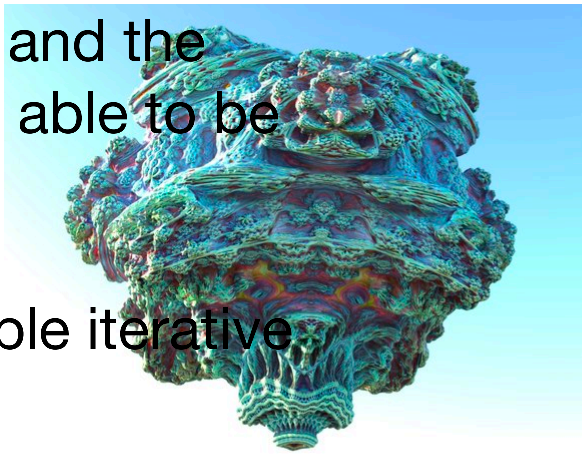
- The map phase applies a function to distributed data;
- The shuffle phase reorganizes the data after the mapping to bring related items together locally
- The reduction phase is typically a summary computation, or merely a data reduction.



Agents and Repositories: the agents and repository pattern is used to solve problems organized as a collection of data elements that are modified at irregular times by a flexible set of distinct operations executed by agents.

The solution is to structure the computation in terms of a centrally managed data repository, a collection of autonomous agents that operate upon the data, and a manager that schedules and coordinates the agents' access to the reposito and enforces consistency of updates.

Iterative refinement: the iterative refinement pattern is used to solve a class of problems where a set of operations are applied over and over to a system until a predefined goal is realized or constraint is met. The number of applications of the operation in question may not be predefined, and the number of iterations through the loop may not be able to be statically determined.



The solution to these problems is to wrap a flexible iterative framework around the operations as follows:

- The iterative computation is performed
- The results are checked against a against a termination condition;
- The computation completes or proceeds to the next iteration.
- Fractal patterns and related images are often carried out by an iterative refinement process.