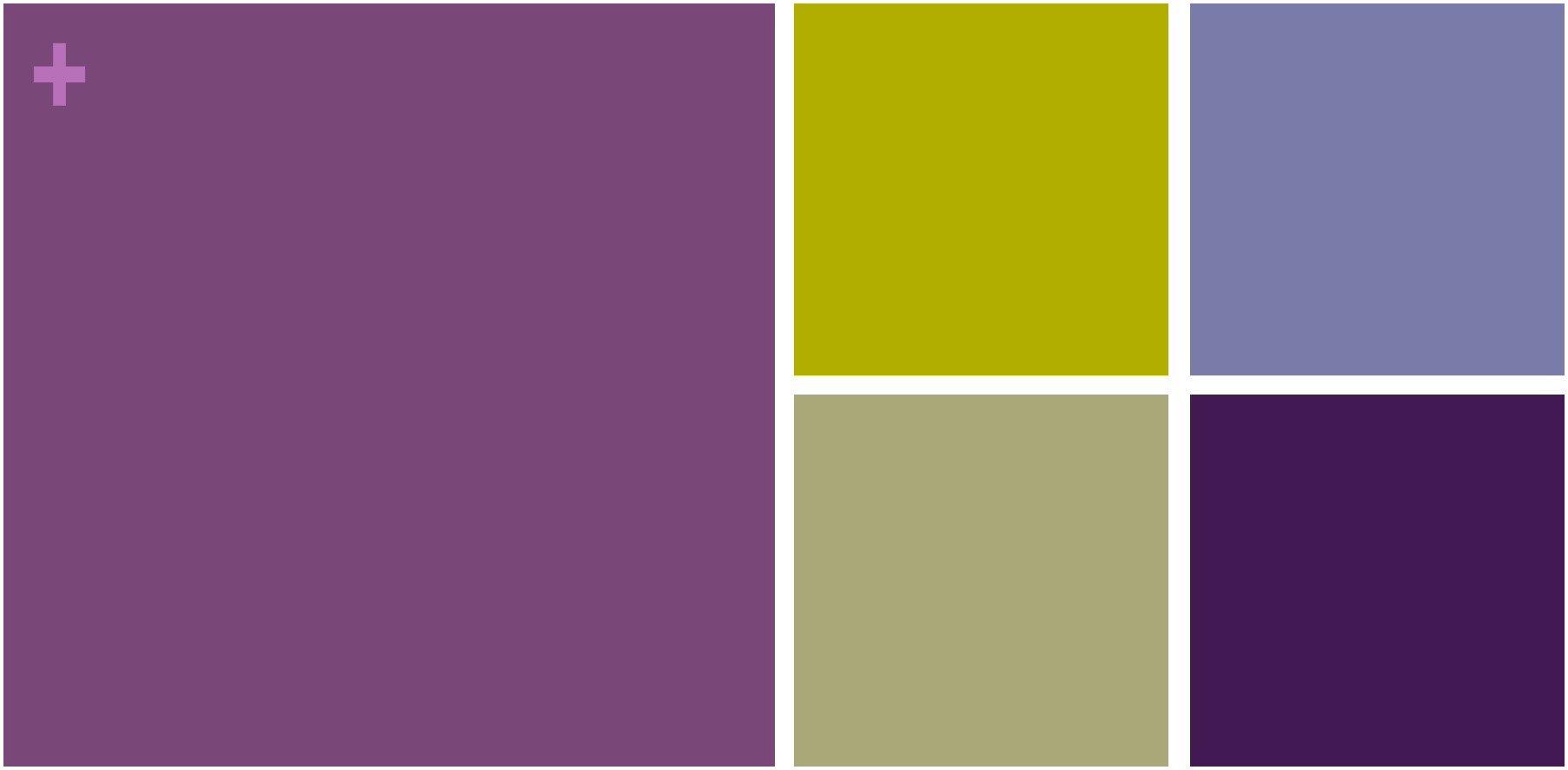


Design of Parallel Algorithms

Introductory Concepts

Parallel Algorithms v.s. Distributed Algorithms

- Distributed Algorithms
  - Focus on coordinating distributed resources
    - Example, ATM transaction processing, Internet Services
    - Hardware Inherently Unreliable
    - Fundamentally asynchronous in nature
  - Goals
    1. Reliability, Data Consistency, Throughput (many transactions per second)
    2. Speedup (faster transactions)

- Parallel Algorithms
  - Focus on performance (turn-around time)
    - Hardware is inherently reliable and centralized (scale makes this challenging)
    - Usually synchronous in nature
  - Goals
    1. Speedup (faster transactions)
    2. Reliability, Data Consistency, Throughput (many transactions per second)

- Both topics have same concerns but different priorities

Parallel Computing Economic Motivations

- Time is Money
  - Turn-around often means opportunities. A faster simulation means faster design which can translate to first-mover advantage. Generally, we value faster turn around times and are willing to pay for it with larger, more parallel, computers (to a point.)
- Scale is Money
  - Usually we can get better and more reliable answers if we use larger data sets. In simulation, larger simulations are often more accurate. Accuracy is be needed to get the right answer (to a point.)
  - Beyond a point it can be difficult to increase the memory of a single processor, whereas in parallel systems usually memory is increased by adding processors. Thus we often will use parallel processors to solve larger problems.
- Analysis of parallel solutions often requires understanding the value of the benefit (reduced turn-around-time, larger problem sizes) versus the cost (larger clusters)

Parallel Performance Metrics

- In parallel algorithm analysis we use work (expressed as minimum number of operations to perform an algorithm) instead of problem size as the independent variable of analysis. If the serial processor runs at a fixed rate of k operations per second, then running time can be expressed in terms of work: $t_1 = W / k$
- Speedup: How much faster is the parallel algorithm:

$$S = \frac{t_1(W)}{t_p(W, p)}$$

- Ideal Speedup: How much faster if we truly have p independent instruction streams assuming *k* instructions per second per stream?

$$S_{ideal} = \frac{t_1}{t_p} = \frac{W / k}{W / (kp)} = p$$

Algorithm selection efficiency and Actual Speedup

- Optimal serial algorithms are often difficult to parallelize
  - Often algorithms will make use of information from previous steps in order to make better decisions in current steps
  - Depending on information from previous steps increases the dependencies between operations in the algorithm. These dependencies can prevent concurrent execution of threads in the program.
  - Many times a sub-optimal serial algorithm is selected for parallel implementation to facilitate the identification of more concurrent threads

- Actual speedup compares the parallel algorithm running time to the best serial algorithm:

$$S_{actual} = \frac{t_{best}}{t_p} = \frac{t_1}{t_p}\frac{t_{best}}{t_1} = SE_a$$

  - Algorithm selection efficiency, $E_a = \frac{t_{best}}{t_1}$, describes efficiency loss due to algorithm used.

Parallel Efficiency

- Parallel efficiency measures the performance loss associated with parallel execution. Basically it is a measure of how much we missed the ideal speedup:

$$E_p = \frac{S}{S_{ideal}} = \frac{S}{p} = \frac{t_1}{p t_p}$$

- We can now rewrite the actual speedup measurement as ideal speedup multiplied by performance losses due to algorithm selection and parallel execution overheads

$$S_{actual} = S_{ideal} \times E_p \times E_a$$

- Note: Speedup is a measure of performance while efficiency is a measure of utilization and often play contradictory roles. The best serial algorithm has an efficiency of 100%, but lower efficiency parallel algorithms can have better speedup but with less perfect utilization of CPU resources.

Parallel Efficiency and Economic Cost

- Parallel Efficiency Definition:

$$E_p = \frac{S}{S_{ideal}} = \frac{S}{p} = \frac{t_1}{p t_p}$$

- "Time is Money" view:
  - Time a process spends on a problem represents an opportunity cost where that is time that the processor can't be used for another problem. E.g. any time a processor spends allocated to one problem is permanently lost to another. Thus

$$E_p = \frac{t_1}{p t_p} = \frac{\$_1}{p \$_p} = \frac{Serial\,Cost}{Total\,Parallel\,Cost} = \frac{C_s}{C_p}$$

- Thus, parallel efficiency can be thought of as a ratio of costs. A parallel efficiency of 50% implies that the solution was twice as costly as a serial solution. Is it worth it? It depends on the problem.

- Note: This is a simplified view of cost. For example, a large parallel cluster may share some resources such as disks saving money while also adding facility, personnel, and other costs. Actual cost may be difficult to model.

Superlinear Speedup

- Superlinear speedup is a term used for the case when the parallel speedup of an application exceeds ideal speedup
  - Superlinear speedup is not generally possible if processing elements execute operations at a fixed rate
- Modern processors execute an variable rate due to the complexity of the architecture (primarily due to small fast memories called cache)
  - A parallel architecture generally has more aggregate cache memory than a serial processor with the same main memory size, thus it is easier to get a faster computation rate from processors when executing in parallel
  - Generally a smartly designed serial algorithm that is optimized for cache can negate most effects of superlinear speedup. Therefore, superlinear speedup is usually an indication of a suboptimal serial implementation rather than a superior parallel implementation
- Even without cache effects, superlinear speedup can sometimes be observed in searching problems for specific cases, however for every superlinear case, there will be a similar case with similarly sublinear outcome: Average case analysis would not see this.

Bag-of-Tasks
A simple model of parallelization

- A bag is a data-structure that represents an unordered collection of items
- A bag-of-tasks is an unordered collection of tasks. Being unordered generally means that the tasks are independent of one another (no data is shared between tasks)
  - Data sharing usually creates ordering between tasks through task dependencies: tasks are ordered to accommodate flow of information between tasks.
- Most algorithms have task dependencies. However at some level an algorithm can be subdivided into steps where groups of tasks can be executed in any order
- Exploiting flexibility of task ordering is a primary methodology for parallelization
  - If an algorithm does not depend on task ordering, why not perform tasks at the same time on a parallel processor…

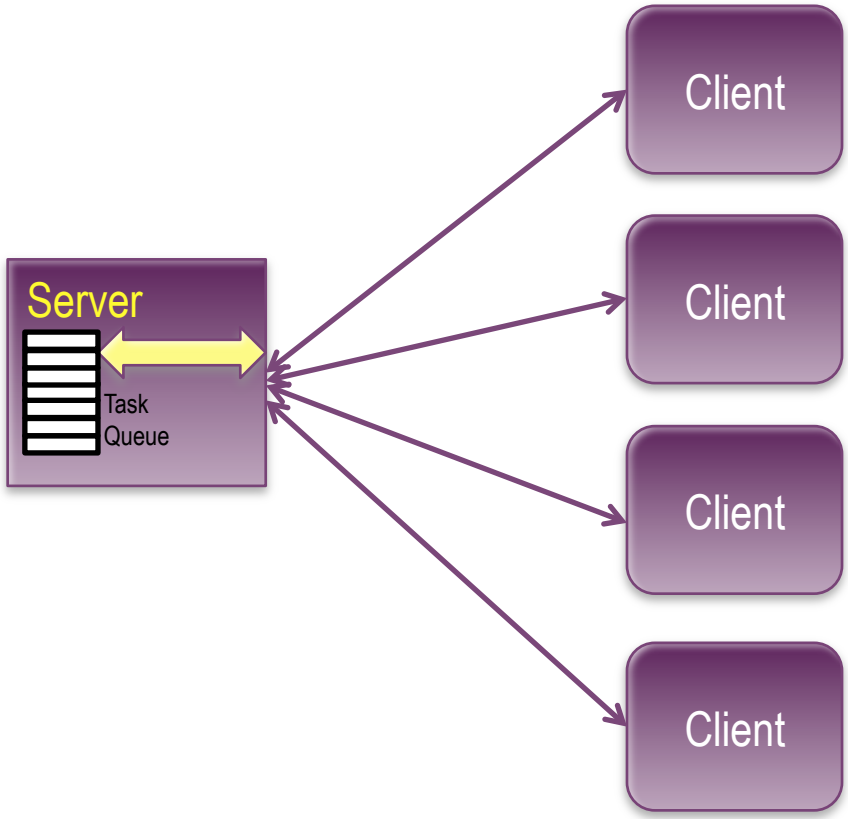
Bag-of-Tasks
A model of parallel program design?

- Generally a task *k* provides an algorithmic transformation of some input data set of size I_k into some result data set of size R_k. In order to perform this transformation, the task will perform some number of operations, W_k.

- If $I_k+R_k << W_k$ then we can can ignore the issue of how the initial input data is mapped to processors (the time to send the data will be much less than the time to compute) and the problem reduces to simply allocating work (task operations) to processors.

- Computations that fit this model are well suited to a bag-of-tasks model of parallelism
  - Note that most computations do not fit this example
    - Dependencies usually exist between tasks invalidating the bag assumptions
    - usually $I_k+R_k \sim W_k$
  - Some examples that do utilize bag-of-tasks model come from peer-to-peer computing
    - SETI at home
    - Folding at home

Implementation of bag-of-tasks parallelism using server-client structure



- Server manages task queue
- Clients send request for work to server whenever they become idle
- Server responds to request by assigning a task from the bag to the client
- Server keeps track of which client has which task so that I_k and R_k can be properly associated when task completes

Performance analysis of the server/client implementation of bag-of-tasks parallelism

- The server will need require some small number of operations in order to retrieve tasks from the queue and organize input and result data. The total work that the server performs will be denoted W_s.
- Task k will require W_k operations to complete.
- Time to solve the problem on a single processor (assuming time is measured in time to perform an operation) is:

$$t_1 = W_s + \sum_{k \in Tasks} W_k$$

Parallel Performance of the server/client implementation

- Assumption
  - The number of tasks is much greater than the number of processors so that we can assume that the amount of time that processors may idle as the task queue becomes empty is a negligible percentage of the overall running time.
  - The communication between the server and client is instantaneous. This is valid if the task execution time is much longer than the transmission times.

- For the parallel performance, we can assume that the time spent on tasks is uniformly divided among clients since a client will not idle until the task queue empties, thus the parallel running time is:

$$t_p = W_s + \frac{\sum_{k \in Tasks} W_k}{p}$$

Sequential Fraction

- The sequential fraction is the ratio of the number of operations that cannot be executed in parallel (server work) to the total operations required by the algorithm. The serial fraction for this client-server model is given by

$$f = \frac{W_s}{W_s + \sum W_k} = \frac{W_s}{t_1}$$

- Given this we can rewrite the parallel execution time in terms of serial fraction:

$$t_p = \left[f + \left(1 - f\right) / p \right] \times t_1$$

- Now the speedup can be easily expressed as:

$$S = \frac{t_1}{t_p} = \frac{1}{f + \left(1 - f\right) / p}$$

Bounds on Speedup: Amdahl's Law

- What happens if we have a very large number of processors?

$$\lim_{p \to \infty} \left(\frac{1}{f + (1 - f)/p} \right) = \frac{1}{f}$$

- Speedup is bounded not to exceed the reciprocal of the sequential fraction!
  - If the server processing required 10% of the operations, then it is not possible to get a speedup greater than 10.
  - Utilizing very large numbers of processors effectively will require very low sequential fractions.
  - Effectively every component of an algorithm will need to be executed in parallel to achieve good scalability on hundreds or thousands of processors!

Real system effects not considered in the present analysis

- Client processors may become idle if waiting on responses from an overloaded server or if task workloads have significant variability. This means that dividing the client work by the number of processors is not accurate.
- Communication times between client and server may be significant. If so, this can be viewed as an increased time spent by the server which will have the effect of increasing the serial fraction.
- The server processor may have a bandwidth limitation, e.g. it may only receive a certain number of requests per second. For large numbers of clients this rate can easily be exceeded. A proper analysis would need a second check to make sure basic bandwidth limitations were not exceeded.