1 Introduction to Parallel Computing

1.1 Goals of Parallel vs Distributed Computing

Distributed computing, commonly represented by distributed services such as the world wide web, is one form of computational paradigms that is similar but slightly different from parallel computing. While the primary goal of parallel computing is to reduce the turn-around time for computing problems, distributed computing has a stronger emphasis on managing distributed resources, security, and reliability in the face of distributed component failures. However, this course is on parallel computing, and so performance plays a key role, while the issues of primary importance to distributed computing remain important but subservient to the fundamental goal of algorithm performance and efficiency.

1.2 An Introduction to Performance Metrics for Parallel Computing

The most fundamental metric of performance in parallel algorithms is speedup, or the ratio of sequential ($p = 1$) execution time to parallel processing execution time,

$$S = \frac{t_1}{t_p}. \tag{1}$$

If we assume that we have a parallel processor that is composed of a collection of serial processors capable of processing a stream of instructions at rate k instructions per second then we can execute $\mathcal{W}$ instructions on a single serial processor in $t_1 = \frac{\mathcal{W}}{k}$ seconds. Similarly, if we assume that the instruction stream is independent of temporal ordering constraints, then we can divide this instruction stream evenly between p serial processors to obtain the same result $t_p = \frac{\mathcal{W}}{kp}$ seconds. This gives us our theoretical ideal speedup (also called 'linear speedup') as

$$S_{ideal} = \frac{t_1}{t_p} = \frac{\mathcal{W}/k}{\mathcal{W}/(kp)} = p. \tag{2}$$

In theory this is an upper bound on parallel speedup since greater speedups would violate our assumption that each processor instruction rate is given by k. In practice, it is possible to exceed this speedup under circumstances where the processing rate, k, is a function of the character of the instruction stream, such as in cache based architectures. Generally these 'super-linear' speedups can be avoided if careful choices are made in the selection of the serial algorithm timings, t_1. For example, if the instruction stream of the serial algorithm is blocked into segments to make better use of cache, then the resulting serial timing measurements would improve resulting in less optimistic speedup measurements.

The problem of choosing the serial time in speedup calculations is made more difficult by the common fact that algorithms with the best potential for parallel execution often are not the best sequential algorithms. Thus, it is common for the time required for a given parallel algorithm executing on one processor to be larger than the best serial algorithm time. Thus, a honest accounting of speedup in parallel algorithms should use the best sequential running time. Using this observation we can define an actual speedup given by

$$S_{actual} = \frac{t_{best}}{t_p} = \frac{t_1}{t_p}\frac{t_{best}}{t_1} = SE_a, \tag{3}$$

where E_a is an efficiency factor due to algorithm selection that is bounded such that $0 \leq E_a \leq 1$.

Parallel Efficiency, E_p, provides a measure of the performance loss associated with parallel execution. Parallel efficiency is a measure of the fraction of ideal speedup actually obtained as given by

$$E_p = \frac{S}{S_{ideal}} = \frac{S}{p}. \tag{4}$$

Now the actual speedup can be expressed in terms of the ideal speedup and the efficiencies of the parallel algorithm as in

$$S_{actual} = S_{ideal} \times E_p \times E_a, \tag{5}$$

where the overall efficiency of the algorithm, $E = E_a \times E_p$, is simply composed of contributions from the choice of algorithm, E_a, and intrinsic parallel (in)efficiencies, E_p.

It is important to note that speedup, a measure of performance, and efficiency, a measure of utilization, often play contradictory roles: for example, maximum efficiency is obtained for the best sequential algorithm, which simultaneously achieves the poor performance of unity speedup. On the other hand, once efficiency falls below $\frac{1}{p}$ we can do no better than unity speedup. In the end, parallel algorithm design represents a careful balance between performance (speedup) and utilization (efficiency).

2 Bag of Tasks

In parallel algorithms we would like to transform a single instruction stream into multiple instruction streams executing on independent processing units. Although this would seem straightforward at first glance, the dependence of instructions on results of previous computations hinder the ability to distribute instructions to processors. Parallel algorithms are distinguished from their sequential counterparts in that they include some form of description that either describes or facilitates this distribution. However, there is no single approach to describing or even conceiving of parallel algorithms, instead there are a broad class of perspectives or design patterns that work well to solve particular classes of problems, but fail in the general case. At the core of these parallel algorithm design patterns are different strategies for partitioning the instruction stream (and associated resources) for allocation to processors. The first and perhaps simplest of these patterns is bag-of-tasks.

The bag-of-tasks pattern is applicable in cases where algorithms can be decomposed into many independent sub-programs. For example, applications involving brute-force searches such as involved in the SETI project or code breaking exercises. Bag of tasks is also useful for parametric studies or for fitness function evaluation in genetic algorithms. A common theme in all of these applications is that a dominant portion of the work can be decomposed into tasks that have execution times significantly greater than the the time required to transfer task data.

The general architecture of a bag-of-tasks approach consists of having a central task server as illustrated in figure 1. This server manages a queue of tasks which are distributed to processors as they become available. In addition to serving tasks to client processors, the server is responsible for integrating the various task results into the final solution. We will call the number of instructions that the server must execute to create tasks and assemble the results $\mathcal{W}_s$.

Tasks are independent, meaning they do not communicate with other tasks. As such, a task can be described by initial data that is transfered to the client processor. When a task completes it transmits its results back to the server processor for final processing. For a task labeled k, we have an initial data packet of size I_k, an amount of work, $\mathcal{W}_k$, representing the amount of time required to complete task k, and finally a result data packet of size R_k as illustrated in figure 2.

2.1 Analysis of the Bag-Of-Tasks Pattern

What is the single processor time for the bag-of-tasks algorithm? For a single processor the server could queue all tasks to itself, resulting in an execution time

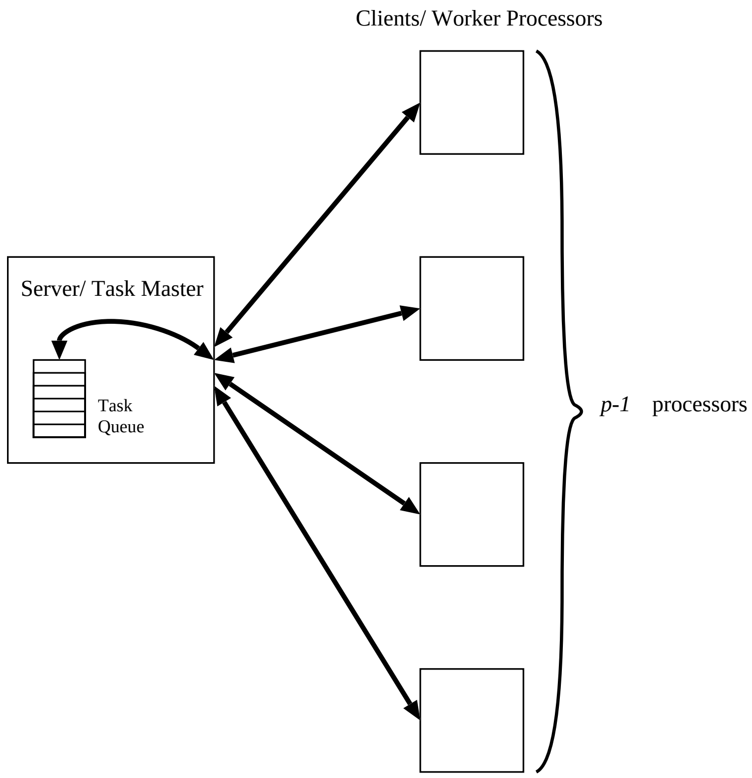

Figure 1: Control model for centralized bag of tasks

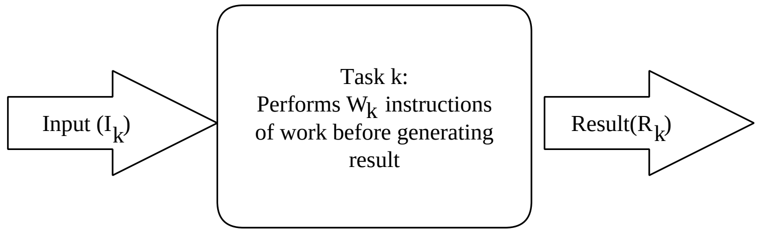

Figure 2: A Task is an independent computational subroutine.

given by

$$t_1 = \mathcal{W}_s + \sum_{k \in Tasks} \mathcal{W}_k, \tag{6}$$

where time is measured in instruction cycles.

If we assume that the number of tasks is much greater than the number of processors and that the communication times between server and client are negligible then the execution time on p processors can be given by

$$t_p = \mathcal{W}_s + \left(\sum_{k \in Tasks} \mathcal{W}_k \right) / p. \tag{7}$$

Note that we can rewrite this equation if we observe that we can write a sequential fraction as the fraction of the sequential work (assigned to the server) to the total overall work as in

$$f = \frac{\mathcal{W}_s}{\mathcal{W}_s + \sum_{k \in Tasks} \mathcal{W}_k}. \tag{8}$$

Then the parallel execution time is given by

$$t_p = (f + (1 - f)/p) \times t_1. \tag{9}$$

This simple model of performance evaluation based on the sequential fraction of a program is called Amdahl's Law. The most fundamental result of Amdahl's law is a relationship between an upper bound on speedup given by the sequential fraction. It can be observed that in the limit as p grows arbitrarily large, t_p approaches $f \times t_1$ bounding speedup to

$$S \leq 1/f. \tag{10}$$

For example, if 10% of the work of a problem is performed by the server, it is impossible to achieve a speedup greater than 10. Thus this illustrates the fundamental limitation of the bag-of-tasks pattern: it is susceptible to server bottlenecks, and can only be effective if the work the server does (including sending and receiving messages) is only a very small fraction of the total amount of work.