Abstract

A parallel concurrent application runs most efficiently and quickly when the workload is distributed evenly among the available processors in a parallel system. This paper gives an overview of some of the issues, concepts and types of load balancing used. The type of balancing can mainly be classified as static balancing or dynamic balancing. Dynamic balancing can be further broken down into various different flavors, all of which are dependent upon the specific application being run and its characteristics. Some methods of dynamic balancing are discussed here. A different type of balancing is based on data locality. This is also discussed here.

Introduction

Parallel computing systems can be loosely broken up into two types, job specific systems and general purpose systems. Job specific systems are highly specialized for the type of job run and general purpose systems are very generic to support a large variety of potential applications. In between these two extremes exists hybrids combining the characteristics of the two.

Job specific systems tend to run on dedicated hardware with strict, pre-defined behavior. The software is generally designed and written to run on one of the single dedicated processors. I tend to view these systems as multi processor embedded real time systems. Each processor in the system has specific functionality and has a predictable effect to itself and the system as a whole. Speaking on industry experience, an example might be a digital radio that processes a received data stream serially. Each processor is responsible for processing a particular aspect of the data stream. One processor is responsible for filtering the stream, a second processor is responsible for decompression, a third processor is responsible for audio conditioning, a forth is responsible for overseeing the actions of all above processors, etc.

This paper will concentrate on general purpose systems. General purpose systems are generic in nature. They lend themselves to process all different types of applications, many of which are not pre-defined. Typically speaking, these systems consist of many homogeneous or heterogeneous processing nodes, connected by some sort of homogeneous network. The hardware is set up to process all different type of applications. There is an operating system that coordinates activity between the various nodes, and can be assumed to be invisible to the software that runs under it. Message passing is used as the underlying form of communication between the processing nodes. These nodes may also share memory between them. I tend to view these systems as networks of workstations. These systems may be used by several users at a time as a time sharing system or they may be dedicated to a single user. In any event, they are set up to run all different types of user defined applications, each different in nature and processing needs. Speaking on academic experience, an example might be a cluster of
workstations tied together by an operating system that supports parallel processing, such as PVM. A user may harness the power of several workstations to solve complex mathematical simulations in less time than would be possible on a standard single workstation.

These applications need to be balanced properly on the system to run efficiently and quickly. The degree of concurrency in the application chosen must not be too coarse. We would like the application to take full advantage of all processing power available. For example if we have P processors and T tasks, generally speaking we would like to have T > P so all processors are busy at some time. If P > T, some available processors would be idle and the system as a whole underutilized. On the flip side, the degree of concurrency chosen must not be too fine. As the number of tasks T becomes large relative to P, generally speaking, the amount of synchronization and communication required to support the application also increases. This tends to bring efficiency down since the time required to support communication and synchronization may eventually exceed the amount of time the system is doing actual work. The degree of concurrency must be held somewhere in between to achieve full utilization of the system. We would like to utilize all available processors to shorten the execution time of the application (figure 1).

![Diagram of load balance]

**figure 1** - Execution time of an a) uneven load balance and b) even load balance

A perfectly balanced system should, in theory, process more work faster than an unbalanced system.

The workload of parallel systems can be balanced either statically or dynamically. These two types of balancing are presented in the following sections. Particular detail is given to dynamic load balancing. Some general observations of balancing an application are presented. Also, some different types of dynamic allocation strategies are explored.
Static Load Balancing

Static load balancing schemes predetermine the individual processor loads prior to runtime. Static load balancing lends itself best to applications that will be run on dedicated systems with predefined characteristics. The various tasks are usually partitioned accordingly at compile/link time. The assignment of tasks may be influenced by the problem size, number of processors available, relative performance of individual processors, etc. However, once the assignment is made, it is not changed at runtime. This is the simplest scheme of load balancing and is usually done by mapping tasks to processors. Given an application with graph A and parallel host system with graph H, the goal is to efficiently embed graph A onto graph H (9). The better mappings tend to reduce required synchronization and inter processor communication to let the application run quickly and smoothly, fully taking advantage of all available computing power.

This type of balancing has the advantage of requiring no runtime support and the overhead that goes along with it, since it is by definition predefined. It also lends itself well to hand tweaking. Past performance may be used as a guide to improve future performance. Static balancing allows a user the ultimate control to predefine all of an applications characteristics.

Although it is the simplest balancing scheme of the two, since nothing is hidden from the user, it generally cannot not take into account such things as processor loading due to other users and higher priority jobs running on the system. It also does not take into account the inherent communication and synchronization delays of the system. Although it can be attempted, it is generally very difficult to predict the runtime characteristics of an application and its tasks without actually executing the task. Thus an application that may seem to be perfectly balanced on paper, may actually be very unbalanced, inefficient and slow. Generally speaking, static load balancing is simple and easily understandable in its nature but practically very limited.

Dynamic Load Balancing

Dynamic load balancing is a method of load balancing that is done dynamically at runtime. Tasks are spread evenly across the available processors and the workload is adjusted accordingly. Dynamic load balancing lends itself to most parallel applications. This type of balancing has the advantage of being customized to the particular system it is being run on at any given time. In a dynamically balanced system, if one processor has more computing power than a second, the balancing algorithm can assign more of the workload to the first processor, fully taking advantage of the processor with the greater horsepower. Some methods may use past execution results to predict future patterns and adjust accordingly. Other methods may use real time performance characteristics extracted from the system at runtime to adjust the application
accordingly. The type of algorithm used to balance the system specifies what balancing mechanisms are used.

One of the obvious disadvantages to dynamic load balancing is the added runtime support required to run the balancing algorithm. Typically, the algorithms monitor, exchange information among processes, calculate workloads and distribute, and in some cases redistribute, the workload (3). Thus, an algorithm that perfectly balances the workload may have the disadvantage that it itself takes too many cycles to run and effectively takes cycles away from the application. However generally speaking, the added overhead of a properly chosen algorithm can be offset by the improved performance of the dynamically allocated and optimized application.

Generally speaking, dynamic load balancing schemes can be classified as one of two types, centralized or distributed (2). Centralized schemes have a central location where tasks are handed out (figure 2). The total amount of work that needs to be done is allocated to individual tasks which are placed into a single work pool. The balancing process initially hands out tasks to all available processors. As a processor completes its assigned tasks, the balancing process assigns new tasks for the processor to run. It operates in a clear master-slave fashion. This type of balancing has the advantage that control is kept in a central location which may be easily monitored. However the scalability of a centralized system is limited. As the number of available processors increases, the amount of communication and synchronization required to maintain the system also increases. A bottleneck may be formed at the distributing process, leading to idle processors waiting for work tasks.

Distributed schemes usually have several load balancing processors. Initially, the work is distributed from the main work pool by a master to several other mini masters that each distribute the work to a cluster of processors below them. These additional mini masters are responsible for their share of the work pool (figure 3). Each mini master runs a load balancing algorithm to manage the work being done on the processors below them. They may be responsible for requesting more work from the master if all total work has not been allocated.
They may also be responsible giving work back to the master for reallocation if their individual load is too high. Distributed schemes have the advantage over centralized schemes of being scalable. The bottleneck seen in centralized systems can be avoided by sharing the load balancing work among mini masters as the number of available processors increases. However, this is accomplished at the expense of additional synchronization, communication and algorithmic complexity.

![Figure 3 - Distributed dynamic load balancing](image)

When balancing a system dynamically, one must also consider the effect of moving tasks between processors. When moving a task from one processor to another based on load imbalances, consideration must be also given to the effect of the communication and synchronization required to actually move the task. For example, if a large amount of data must be also be moved to migrate a task from a busy processor to an idle processor, the involved communication and synchronization time may actually outweigh the benefits of moving the task in the first place.

As one can imagine there are several methods of dynamically balancing a system. The best one is of course, dependent on the type of application, its tasks and the underlying system on which it is being run. Some methods of dynamic balancing are detailed below.

**A Hybrid Centralized vs. Distributed Balancing Scheme Taking Into Account Global vs. Local Information Exchange**

As presented in (3), there exists a hybrid model that not only takes into account central vs. distributed control, but also global vs. local information exchange. In global schemes, all processors used broadcast their performance profiles to all other processors. In local schemes, processors are grouped into clusters. These clusters share their performance profiles locally among each other. This hybrid model uses the two variations to produce four different models - Global Central, Global Distributed, Local Central and Local Distributed.
In the Global Centralized model, the load balancer is located on a central processor. The performance profiles are broadcast throughout the system. The balancer calculates the new work load distribution and sends individual instructions to each processor to control load balancing. In the Global Distributed model, the load balancer is replicated on all processors. The performance profiles are still broadcast throughout the system. However, Individual instructions are not required to control load balancing since all balancers have the required information needed to rebalance the system.

In the Local Centralized model and the Local Distributed model, processors are partitioned into different groups of size K. The local schemes are the same as their global counterparts, except for the fact that performance profile information is exchanged only within the local group, which as a whole reduces the amount of communication in the system compared to global schemes.

When comparing the global vs. local schemes, the global schemes tend to converge faster than the local schemes, since the performance information is available at a common synchronization point. However, it is done so at the added cost of communication and synchronization. For the local case, any differences in computing power between the clusters may have a significant impact. One cluster may sit idle waiting for the other cluster to finish.

When comparing the central vs. distributed schemes, as pointed out above, the central point of control can cause bottlenecks and limit scalability. Again, by distributing the balancers, the scalability can be increased at the added communication and synchronization expense.

Using these models, the authors of (3) compared performance evaluations of the matrix multiplication $Z = X \times Y$. The rows of $Z$ and $X$ were distributed and $Y$ was replicated on each processor. When work was redistributed, only the rows of $X$ needed to be communicated. Generally speaking, the performance observed was better using local schemes than global schemes. This is most likely due to the fact that the communication required is less in the local case and the simplicity and uniformity of the problem itself. The advantages of distributed control starts to be seen as the number of processors used increases.

**Adaptive Collision Protocol**

A dynamic load balancing model is described in (4). This model uses a number of balls $M_i$ to represent tasks and a number of bins $N_j$ to represent available processors. The goal is to place the balls into the bins most efficiently to come up with the fastest execution time. Furthermore, the balls were given weights $W_i$ to simulate variable sized tasks. The model described is based on centralized load distribution. It can be easily extended to work as a distributed load balancing scheme by placing additional balancers below the master node running the same or similar algorithms.
This scheme is different from simple work pool queue models in that it attempts to balance the load based on job size. Work pool queue models tend to feed available processors with the next task in the queue, regardless of the size of the task. Although the Adaptive Collision Protocol balances the workload based on task size, it does so at the expense of more communication and higher algorithm complexity. Simple work pool queue models do away with the algorithm complexity.

The Adaptive Collision Protocol model described does not take into account different size bins, i.e. processors with different performance characteristics. A further extension of this model may be to place weights on the bin sizes to simulate processors with different performance capabilities. Again, this is done so at the added expense of higher algorithm complexity.

**Virtual Data Space**

Virtual Data Space (VDS) is a standard C library that automatically distributes tasks generated by parallel applications to the available processing nodes. Since it is a library, the support it provides is used by placing VDS primitives within an application.

In order to distinguish between different types of tasks or objects, VDS uses the C++ class concept to group tasks. Each task is declared by the user in the code to be of a certain VDS predefined class. At runtime, tasks are placed into a global space where they can be distributed to available processors. The user may specify which types of classes each processor may operate on. Thus it is easy to restrict access to some tasks by defined processors and thus run groups of related tasks on the same processors. The scheme of balancing can either be qualitative or quantitative based on the class definitions of an applications tasks.

Although VDS is not really a balancing algorithm as described thus far, it is still useful in that its use can dynamically balance an application. It does give the user some static control over the balancing by the class definition of tasks. The actual balancing however is done at runtime. Note by specifying which tasks get processed where, balancing with respect to memory locality, as described below, can be adhered.

**Load Balancing vs. Memory Locality**

Up to this point we have explored balancing schemes that are determined based on distributing the computational load evenly among processors. In shared memory multiprocessor systems, it is also desired to take into consideration the locality of the data required by different processes. (6) and (7) explore the effects of balancing not only the workload, but also balancing tasks based on the closeness of the data. There is often a conflict when balancing a system based purely on loading or purely on memory locality. Since balancing based on processor loading has been solely discussed up to this point, the effects of memory locality is now considered.
Speaking by example, suppose we have four tasks (T1, T2, T3 & T4) to be run on two processors (P1 & P2). T1 and T2 share common input blocks of data as do T3 and T4. If T1 is allocated to P1 and T3 is allocated to P2 initially, it may be optimal if T2 is run on P1 and T4 is run on P2 even if T1 and T2 are small computation wise relative to T3 and T4.

A typical load balancing scheme such as the Adaptive Collision Protocol may try to allocate T1 and T3 on P1 and T2 and T4 on P4 to balance the computation load. However, when considering memory locality and the contents of the cache in P1 and P2 after executing T1 and T3, respectively, it may be far more beneficial to allocate T1 and T2 to P1 and T3 and T4 to P2. Even though P1 will be idle while P2 is doing work, by allocating the tasks with similar data requirements, we reduce the amount of communication required in a message passing system or the memory access time in a shared memory system to update the data in the cache required by the new tasks. By providing the second tasks access to cache that has been pre-loaded by the first tasks, we effectively remove the time needed to load the cache, and thus lower the overall execution time. This is of course dependent on the size of the data blocks and the time required to move these blocks through the system. Balancing with respect to data locality may be applied both dynamically as well as statically.

It was shown in (6) that Guassian elimination and Shellsort algorithms ran faster when using a data locality balancing scheme over a load balancing scheme. When a mix of the two was applied, the performance seen was closer to the data locality results than the load balancing results.

As with all other balancing schemes, balancing with respect to data locality is highly dependent on the application, its tasks and the underlying system characteristics.

Conclusions

Balancing a concurrent parallel system is dependent upon many related and unrelated factors. Generally speaking, a system may be balanced statically at compile / link time or dynamically at runtime. The balancing may be based on processor loading, a task’s data locality and many other factors. The granularity of the tasks also affects balancing. Too fine a task causes excessive communication to switch tasks in and out of processors. Too coarse a task causes processors to sit idle because there are not enough tasks to keep all processors busy. Too much dynamic balancing may actually degrade a system’s performance due to the communication, synchronization and complexity of the balancing algorithm used. It is also important to keep in mind that when tasks are migrated from one processor to another to satisfy dynamic balancing requirements, an associated communication and synchronization performance hit is also taken to move the task and its data.
There seems to be plenty of research dedicated to the field of balancing concurrent parallel systems. However there is not, nor will there ever be, a single universally accepted and used generic balancing method that achieves the best performance under all circumstances. There are just too many independent factors that effect the performance in these types of systems. To put it another way, “You’re mileage may vary”.

References

4. Petra Berenbrink, Friedhelm Meyer auf der Heide and Klaus Schroder, “Allocating Weighted Jobs in Parallel”, University Paderborn Germany
5. Michal Cierniak, Mohammed Javeed Zaki and Wei Li, “Compile-time Scheduling Algorithms for Heterogeneous Network of Workstations”, University of Rochester
7. Evangelos Markatos and Thomas LeBlanc, “Memory-Conscious Scheduling in Shared-Memory Multiprocessors”, University of Rochester
10. Thomas Decker, “Virtual Data Space - A Universal Load Balancing Scheme”, University of Paderborn Germany
11. Thomas Decker, Markus Fischer, Reinhard Luiling & Stefan Tschoke, “A Distributed Load Balancing Algorithm for Heterogeneous Parallel Computing Systems”, University of Paderborn Germany