ICOT Technical Memorandum: TM-0276

TM-0276

Load Balancing
in
Very Large Scale Multi-Processor Systems

by
T. Chikayama

February, 1987

ICOT
Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191~5
Telex ICOT J32964

Institute for New Generation Computer Technology

Load Balancing
in
Very Large Scale Multi-Processor Systems

Takashi Chikayama
ICOT Research Center

Abstract

An idea for load balancing in very large scale multi-processor systems is presented. The basic idea of the method is to consider the multi-processor system to be an n-dimensional hyper cube in which processing power is uniformly distributed. The application programmers should only be aware of this hyper cube rather than individual processors, and load balancing in this model is to distribute required computation in the hyper cube as uniformly as possible. Load balancing in a lower level can also be done automatically by changing mapping of processors to this hyper cube.

1 Introduction

When considering a very large scale multi-processor system, the ratio of the physical size of the system and the physical size of one processor becomes so large that it cannot be ignored in the system design. When processors should be packed in a 3-dimensional space (which is the only realistic method in this cosmos where we live), the most distant two processors cannot be nearer than $\sqrt{3}d(\sqrt[3]{n}-1)$, where d is the distance between two adjacent processors and n is the number of the component processors. This becomes more than $100d$ for a system with one million processors.

In such systems, an architecture which neglects physical distances between element processors cannot be a *good* architecture. To make all pairs of two processors equally distant, for example, all such pairs must be made as distant as the physically most distant processors. A n-dimensional hyper cube connection with n greater than 3 cannot fit in 3-dimensional physical space. Tree-like structures should not assume all adjacent nodes to have the same distance; adjacent nodes in upper levels of the tree structure cannot but be more distant than those in lower levels to fit in 3-dimensional physical space. If some processors are nearer than other processors, the software might utilize the locality of computation.

This requirement of utilizing the locality constrains feasibility of load balancing methods considerably. If the locality assumed by the software should not be kept by the load balancing procedure, then it is quite likely that the increase of communication overhead cancels all the merits of balancing. Thus, load balancing methods for such systems must not destroy the locality of computation.

2 Model of the Processing Hardware

It might be quite difficult to develop complicated software for very large scale multi-processor systems considering the physical structure of the system directly. A higher level model for processor systems may help software construction very much. The following characteristics are desirable in such a model.

(1) The same model should apply regardless of the number of the processors in the system.

(2) The same application software should be efficient regardless of the number of the processors in the system.

Above items are quite difficult to be realistic without the following assumption.

(3) The software (the programmer and/or the language processor) should be aware of the *locality* of the computation.

The physical structure of the multi processor system cannot but have some inter-processor boundary, but, it is not desirable that the software should be aware of such discrete boundaries (according to the principle 2 above). The locality of the computation, however, must be expressed somehow in the model (principle 3). Thus, the model should have some notion of *contiguous distance*.

This notion of *distance* must have some correspondence with the physical distance in the hardware system. This leads to a quite naïve model which almost directly corresponds to the physical structure of the system — to consider the system to be a hyper-cube in an n-dimensional Euclid space in which computing power is uniformly distributed. To be realistic with our cosmos,

this n must be at most 3. For convenience of explanation, we will assume in what follows that n is 2, and call this 2-dimensional cube (or plane) to be the *Processing Power Plane*, or $\mathbf{P}^3$, in short.

Any computation is located at some point in $\mathbf{P}^3$. The distance of two computations is modeled by the distance of two such points. This also is a not-so-bad approximation of the physical communication overhead.

3 Keeping Locality

To keep the locality of communication, the software must be aware of the distance between two computations. This might be realized by the following way.

- Programs should be organized so that processes requiring more communication fork later. That is, in the process tree structure, processes with more communication have their common ancestor process in lower levels.
- The initial process is given the whole $\mathbf{P}^3$ for its use.
- Each process is given with some sub-rectangle of the rectangle given to its parent.

Using the above allocation mechanism, two processes are always be allocated inside the rectangle area allotted to their latest common ancestors, which gives a certain lower bound of locality.

Another probably feasible method of keeping locality is to allocate processes to where the data required by them are located. When, for example, the generator consists of many processes scattered all around $\mathbf{P}^3$, and each such leaf process generates some substructure of the generated data, then the generated data (probably with some tree-like structure, corresponding to the generator processes tree structure) will also be scattered all around $\mathbf{P}^3$. In such cases, the consumer processes should be allocated using the same strategy as the generator processes, so that they are allocated at the same point on $\mathbf{P}^3$ with the data generated by the corresponding generator process. Of course it might be possible to make one complete process tree of generator-consumer processes in the first place. However, sometimes it would be more convenient to separate them into two-pass style for keeping modularity and/or readability of the program.

4 Load Balancing on $\mathbf{P}^3$

To *logically* balance the computational load, the programmer and/or the language processing system are responsible for balancing on $\mathbf{P}^3$. To achieve this, the programmer and/or the language processor should have at least vague knowledge about how much computation (relatively) is needed for a certain process.

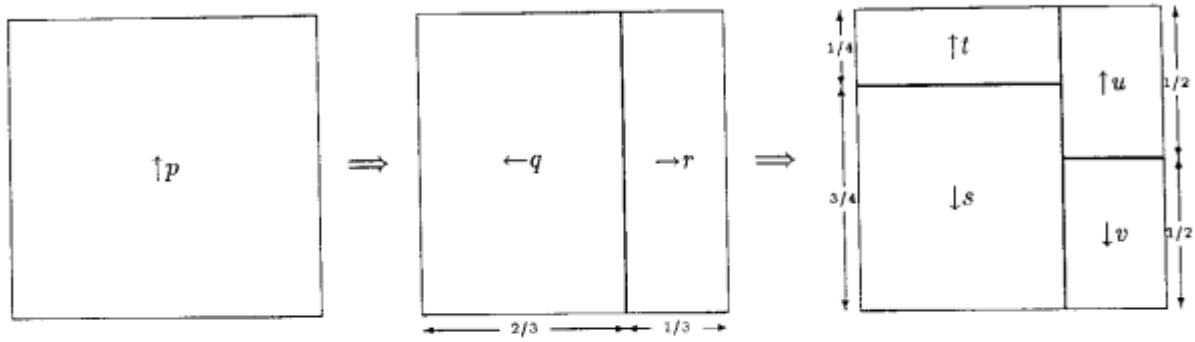
One notation proposed for Prolog-like languages is something like:

$$p :- \leftarrow(2 \times q),\ \rightarrow r.$$

By the above specification, the subplane given to the predicate p will be subdivided for q and r as shown in Figure 1b. This specifies that the subgoal q should be allotted twice as large subplane as the subgoal r.

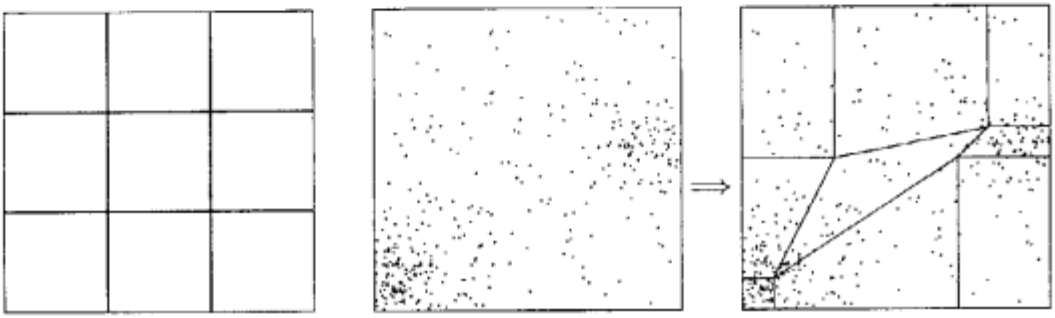
Arrows before the body goals specify which way the subdivision should be when body goals are subdivided again by their reduction in turn. By reductions using clauses such as:

$$q :- \leftarrow(3 \times s),\ \rightarrow t.$$



a. Initial state b. After reducing p c. After reducing q and r

Figure 1: Load Balancing by Subdividing $\mathbf{P}^3$



a. Initial allocation b. Load distribution c. Reallocation

Figure 2: Load Balancing by Reallocation of Processors to $\mathbf{P}^3$

and

$$r :- \leftarrow u,\ \rightarrow v.$$

rectangles for q and r are subdivided again as shown in Figure 1c.

It would be sometimes impossible to specify such load balancing information statically on the source code. It might be possible, however, in certain kinds of programs, to guess amount of computation required at runtime from the given data (sizes of argument structures, for exmple). In such cases, the subdivision parameters can be computed depending on such data.

5 Load Balancing on the Physical Level

$\mathbf{P}^3$ must somehow be covered by the physical processors. When the whole computation starts, all the processors in the system should be responsible for the same amount of $\mathbf{P}^3$ as in Figure 2a. However, as perfect load balancing on $\mathbf{P}^3$ seems to be impossible for complicated algorithms, it is quite likely that some area of $\mathbf{P}^3$ becomes quite dense in computation and some area quite sparse (Figure 2b).

This could be corrected (at least partially) by changing the size of the region in $\mathbf{P}^3$ each processor is responsible for. Processors responsible for dense area should narrower their territories and those for sparse areas should widen them (Figure 2c).

Two strategies may be practical in applying the reallocated processor–$\mathbf{P}^3$ map. One is to reallocate all the processes (probably in the scheduling queue) according to the modified allocation map. Another is to allocate only newly forked processes according to the modified map. The former method is quicker in balancing the load, but requires more inter-processor communication to reallocate goals in the queue. When considering parallel logic programming language such as GHC, there are basically no permanent processes. Goals are always reduced into several descendant goals. Thus, the latter method might be as good.

Assume that a process is allocated quite close to some processor boundary and communicating frequently with another process allocated on the opposite side of the boundary. Though they are almost adjacent on $\mathbf{P}^3$, communication cost with such a process will be much higher than the cost with other processes, not as close on $\mathbf{P}^3$ but are allocated on the same processor. Such processes can be the bottleneck of the whole computation. To avoid such cases, some randomness might be desired in mapping $\mathbf{P}^3$ to physical processors.

Locality is required also in the reallocation. If some centralized controller should be required for such an adjustment, much communication might be required for exchanging load balancing information. Thus, reallocation should be decided locally. A simplest way is by relocating the corner point shared by four adjacent processors, depending on the *loads* of the four processors. This method requires quite local communication only. Computation will be distributed to other processors in a diffusion manner.

What the word "load" means is not quite clear. The number of ready processes in the scheduling queue of a processor is one of the most naïve estimation. One possible method might be to add some load estimation value to each process (by the software) and consider the sum of such values of processes in the scheduling queue to be the load of the processor. Another method might be to try the reallocation only when some processor becomes idle.

6 Requirements for Algorithm Design

It is widely understood that algorithms good for sequential processors may not be as good for parallel processors. With the above processor model, what is important is not only the intrinsic parallelism, but also whether a certain algorithm has good nature in the following view points.

Communication Locality: An algorithm is better when processes require less global communication. Some measurement resembling the notion of working set in sequential programming is required as efficiency criterion. This will be probably more important criterion than working set.

Feasibility of Load Balancing: An algorithm is better when it is easier to predict required amount of computation for each subproblem. Some new measurement is required here again.

The extreme in the above sense are systolic algorithms, where amount of communication and computation required by subproblems are completely known beforehand. As we are interested in more complicated problems, some flexible measures are required.

7 Future Research Plans

The following items must be studied before the $\mathbf{P}^3$ model should be made practical.

Static Locality Analysis: Development of algorithms for automatically extracting locality information from programs where no explicit information is given. As cache or virtual memory systems does work with most of the programs which are written without even considering the existence of them, this study might be fruitful.

Automatic Load Balancing: Development of automatic balancing algorithms and their evaluation. Through this, some knowledge would be gained on what sort and how much of load unbalance the systems designed on the $\mathbf{P}^3$ model are bearable.

Parallel Algorithms: Development of parallel algorithms, considering communication locality and load balancing feasibility. This will probably lead to a set of algorithms quite different from sequential algorithms widely used currently. They might also be quite different from algorithms for parallel execution with equal-distance assumption.

Acknowledgments

The author wants to thank many researchers in and out of ICOT, too numerous to be listed here. Most of the ideas presented in this paper are developed stimulated by discussions with them.