Load Balancing on Massively Parallel Networks: A Cellular Automaton Approach

by Brian Dickens
at the Thomas Jefferson High School for Science and Technology

supervising instructor: Donald Hyatt

originally published December 1992
scanned/retyped into Google Docs in December 2013

for more information, refer to blog post:
http://hostilefork.com/2013/12/10/load-balancing-teenage-coding/

Abstract

For efficient parallel processing, each node in the network must be exploited completely. Dynamic load balancing, which involves the migration of tasks from heavily loaded processors to those with lighter workloads, is a method for achieving this goal. This paper presents a completely distributed algorithm for performing dynamic load balancing. The algorithm is independent of the network structure, and uses a set of cellular automaton rules to apply methods of gaseous diffusion to the process of system partitioning. Effects of varying the parameters of this algorithm are studied using an arbitrary network simulator, and optimal parameter values are determined.

1.0 Introduction

Advances in technical fields constantly require faster computation, but computer hardware manufacturers have reached fundamental physical speed limitations [3]. Advances in electronic technology alone are unable to keep up with this demand for increasing computational speed. The emerging answer to this problem is parallel processing, in which different parts of a task are executed simultaneously on multiple processors [1,2,3].

Although it is generally considered more difficult for programmers to write code that is flexible enough to be divisible among several processors, the push toward parallel processing hardware and software has become increasingly prominent [2]. Rather than limiting a program's execution time by running it on one processor, the workload in parallel processing is divided among several processors, allowing the problem to be solved through "teamwork." Thus, parallel processing has become a viable alternative to faster circuit and packaging techniques, which can merely reduce the basic cycle time of single processors [3].

Despite these theoretical advantages, parallel processing can only be beneficial if resources are effectively managed [1]. If a program is only running on one processor in a network containing thousands of processors, then the other processors will not speed up the program. To achieve high utilization of system resources, the total workload must be divided into independent subsystems, so that several jobs may execute simultaneously [4]. This division of workload is referred to as system partitioning [1].

System partitioning processes can be classified as static and dynamic [1]. In static partitioning, tasks enter the system in a queue, and are assigned to available processors [1,6,7]. This technique distributes the workload independent of job sizes, and once a job is placed on a processor, it cannot move to another [1]. Thus, static partitioning is relatively simple to implement.

In dynamic load balancing, the system is periodically updated, and tasks migrate from one processor to another based on current job sizes [1,4,5]. Although dynamic partitioning is more complicated than static partitioning, it results in improved utilization of a system, since the workload on the network is constantly adjusted [1]. Dynamic partitioning is also known as dynamic load balancing, and is the focus of this project.

At startup in dynamic load balancing, processes are distributed to lightly loaded processors. Later they may be moved to other processors. This movement, referred to as dynamic migration, involves briefly freezing the progress of the unit(s) of migration, moving the migration unit(s) to a destination processor, implementing one of several mechanisms for rebinding references, and finally resuming execution [4].

2.0 Related Work and Motivation for a New Approach

Dynamic load balancing requires the intelligent migration of tasks from heavily loaded to lightly loaded processors. This migration could be controlled by a “master” processor, which gathers global information about the state of the system. Based on the information gathered, the “master” processor would schedule the transfer of tasks between individual nodes, directing work towards idle processors. This centralized approach has been studied in many research papers [1,2,3,4,5].

Although the centralized approach to load balancing has the potential to minimize communications, it can be extremely inefficient [5]. During the time in which the “master” processor is gathering information from the network and scheduling task migrations, the other processors are left idle. Paradoxically, this underutilization of the system’s resources is the exact problem that load balancing algorithms are attempting to avoid. In addition, one can foresee future supercomputers consisting of thousands of processors [5]. With such massively parallel systems, it would be impractical for a central “master” node to have direct communication links connecting it to thousands o f “slave” processing components. Conventional interconnection devices, such as single shared buses or rings, cannot meet such demands [2].

In today’s massively parallel computers, as well as in many local area networks, processors have local connections rather than global ones [2,5]. These processors can communicate directly with their neighbors, but not with any other processors [5]. A centralized algorithm for load balancing is impossible in such a network, since no processor can evaluate the status of the entire network. In fact, each processor has only partial knowledge of the network based on what it can learn from its neighbors, and no single unit can be considered the “master” processor. The hypothesis of this project is that load balancing can be achieved on locally-connected networks using a cellular automaton approach.

Cellular automata are discrete dynamical systems whose behavior is completely specified in terms of a local relation [8]. Originally popularized by Conway's game of "Life", they are analogous to many aspects of parallel processing [8]. In a cellular automaton, each "cell" or "node" makes a decision affecting its state, according to a "rule" involving the state of its neighbors. Every application of the rule results in a new automaton status, referred to as the next "generation." To proceed from one generation to the next, the rule is applied to every node simultaneously, making the principles of cellular automata easily adapted to parallel processing systems.

The cellular automaton used in this simulation implements a new "diffusion" rule. The phenomenon of gaseous diffusion is nature's paradigm for load balancing, since a gas which is originally concentrated at one location will spread evenly throughout its container [9]. In other research, the mathematics behind diffusion have been used as a model to evaluate distribution algorithms [6,7], but this paper suggests that an algorithm modelling diffusion can be applied to load balancing on massively parallel networks.

3.0 A Generic Modelling Environment for Process Distribution

To test the performance of load balancing algorithms, a simulator for theoretical parallel processing systems was written. For this simulation to be practical, a few simplifications had to be made. The first simplifying assumption is that all processors in the network have identical computational capacities, and are connected by links which transfer information at equivalent rates. Since many of today's commercial parallel computers have thousands of identical processors with identical local connections [4], this assumption is realistic.

The other major assumption made in this model is that all work is homogeneous, and can be represented by a floating point number. This number is a relative indicator of the number of computations that a processor has to perform, so that a processor with a work value of one has half as many computations as a processor with a work value of two [5,6,7]. The abstraction clearly applies to independently executable tasks, such as the transformation of pixels in image processing, where each node can perform computations on a data item without requiring the use of information held by other processors. It also applies to multitasking networks, where the floating point number can represent the number of tasks that a particular processor has to switch between.

The arbitrary network simulator was written in the C programming language on a Zenith 386/20 DOS system. In the simulator, a processing element is referred to as a "node", and a summary of the basic routines for managing nodes can be found in NODE.H. These routines provide for the allocation and deallocation of nodes (`makenode` and `killnode`), the establishment and removal of connections between nodes (`makeconn` and `killconn`) as well as means for querying the contents of direct neighbors (`neighbor` and `looklink`).

Each node stores state information indicating how much work it has, how much it is going to transmit to its neighbors, and how much it is going to keep. The values for these variables are determined through the application of cellular automaton rules. In this model of load balancing, the algorithm seeks a state in which each node has a workload equal to the average workload of the entire system.

The load balancing of a simple circular trinary network is shown below:

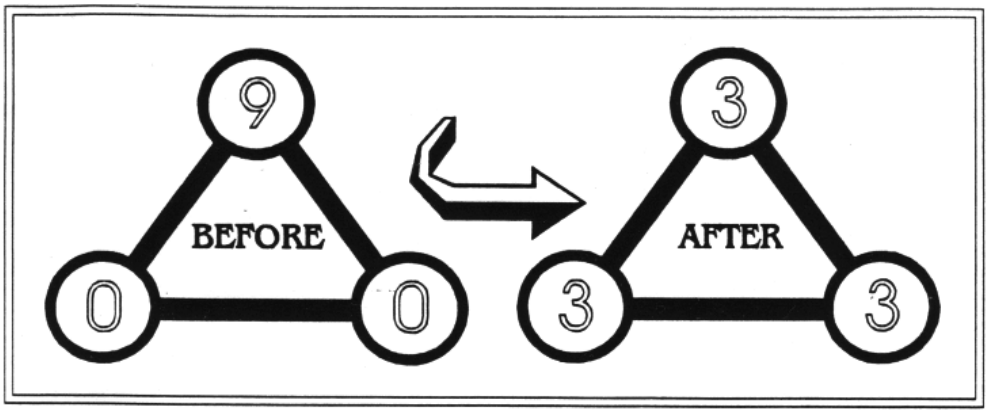

Figure 1: A trinay network, before and after load balancing.

Before load balancing, there is an uneven distribution of work, and the entire workload of the network is concentrated on one node. After load balancing, it can be seen that each processor is bearing its fair share of the workload. The method through which the network proceeds from the “Before” diagram to the “After” diagram is the focus of this project.

3.1 A Diffusion Model

The load balancing model that was developed is based upon an automaton model of gaseous diffusion. The process of gaseous diffusion is dependent on the laws of entropy to bring about an equilibrium state. If there is a high concentration of gas molecules in one area of a container, then those molecules do not “perceive” a deficit in the less concentrated areas , which they then attempt to fill. Likewise, molecules in less concentrated areas do not “perceive” a surplus in the more concentrated areas.

In reality, molecules are constantly traveling throughout the container, regardless of the concentration in their particular region. They spread from the areas of lower concentration into the areas of higher concentration at the same rate at which they spread from the areas of higher concentration into the areas of lower concentration. Although the rates at which molecules are travelling are equal, the quantities transferred between areas of different concentration are not.

This can be demonstrated by a situation where one region of a container contains ten gas molecules, and another contains five. If 1 out of every 5 molecules diffuses in a unit of time, then during one step, 2 molecules will be transferred from the area containing five into the area containing ten, while only one molecule will be transferred the other way. After this diffusion has taken place, the areas will contain nine and six molecules respectively.

It can be seen that since there are more molecules in concentrated areas, the relative quantities of molecules leaving those areas are greater, even if transfers between areas are conducted at equal rates. Eventually, the quantity of molecules leaving any given area becomes equal to the quantity of molecules entering that area, and equilibrium is reached [9]. The application of this principle to cellular automata and parallel processing networks is demonstrated in Figure 2:

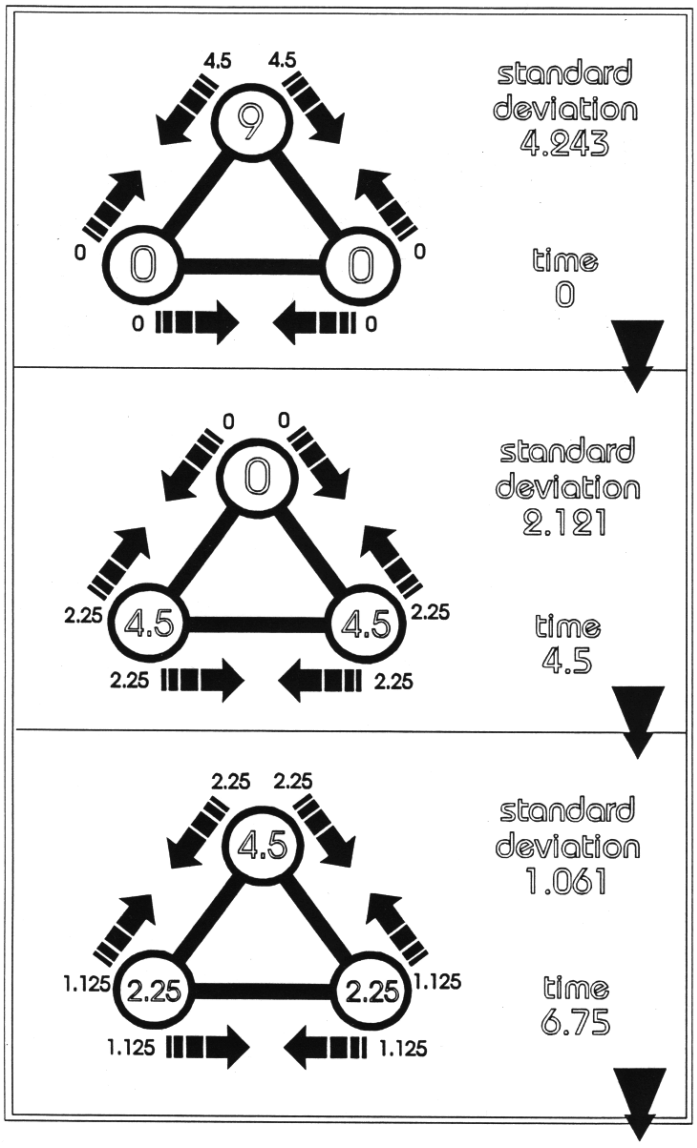

Figure 2: Diffusion approach, where each node sends half its workload to each neighbor.

An unbalanced trinary circular network of processors is given, in which the workload is concentrated on one of the three nodes. In this simple model, each processor distributes work by giving half of its workload to neighboring processors. In the first generation, the loaded processor gives 4.5 work units to each neighbor. The neighbors, having zero workload, give half of zero to each neighbor, thus transferring no work. In the second generation, the processor which was originally loaded no longer has work to give, but receives 2.25 work units from each of its neighbors. In the third generation, it can be seen that each processor has work which it splits between its neighbors.

The process may be continued indefinitely, and if the automaton were carried out for several additional generations, the system would eventually reach an equilibrium state. In such a state, the amount of work leaving each processor would equal the amount of work entering each processor, resulting in a load balanced network.

4.0 Simulation Methodology

In the process of gaseous diffusion, molecules are constantly travelling throughout a container, even after equilibrium is reached. This characteristic is undesirable of the work transfer in a load balancing process, where the number of communications should be as low as possible, to save processing time. In this model, once a balanced network state exists, the diffusion process is halted and computation is initiated. A related problem is represented below:

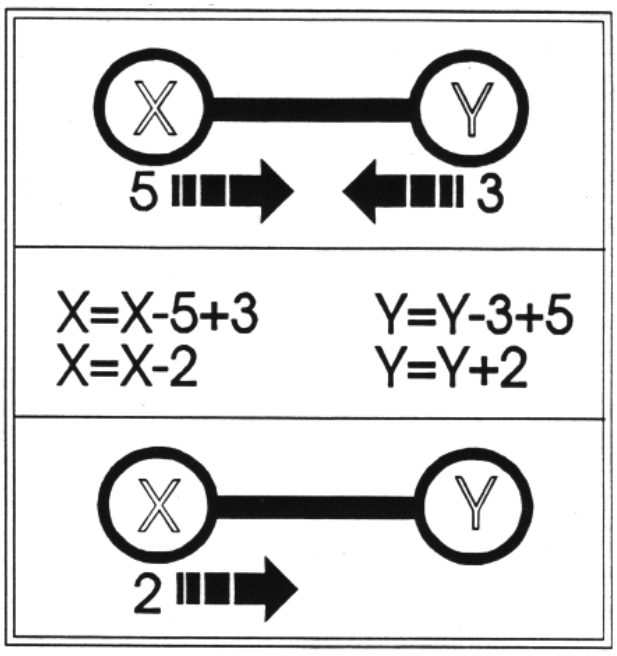

Figure 3: Minimization of data transfer in the diffusion-based load balancing algorithm.

Since all work is considered to be homogeneous, there is no practical advantage to transferring three units of work from processor Y to processor X, only to have all three units replaced. The top example would require five communication time units, while the reduced transmission algorithm shown at the bottom of Figure 3 requires only two communication time units. This technique of sending only the load transmission difference between processors is implemented in the arbitrary network simulator, to minimize communication time.

Gaseous diffusion is guided by the shape of the container, and not by the specific concentrations in areas inside the container. In much the same way, diffusion on a network should be determined by the network's topology alone, and not the load state of the network. As a result, each processor in the diffusion cellular automaton must decide what proportion of its workload to send to its neighbors on the basis of its neighbors' network connections, and not their relative work values. For diffusion to successfully balance a network, there must be some rate at which work is transferred between processors. In addition, the proportion of work which a processor gives away must equal the proportion of work which it receives from its neighbors.

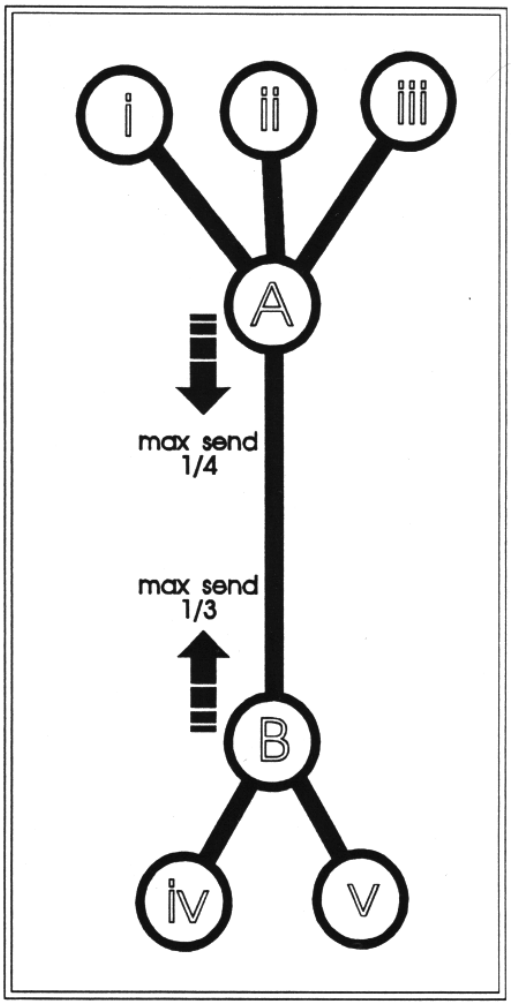

Figure 4: The need for an agreement between adjacent processors on a diffusion rule.

Assuming that all neighbors are being considered equally, processor A knows that since processor B has three neighbors, it can offer to distribute a maximum of 1/3 of its workload. Conversely, processor B knows that processor A has four neighbors, and can donate a maximum of 1/4 of its workload. If both processors attempt to send their maximum proportions the status of processors A and B after one generation would be:

$$A = A - (1/4\ A) + (1/3\ B) \qquad B = B - (1/3\ B) + (1/4\ A)$$

These equations make the assumptions that processors **i**, **ii**, **iii**, **iv** and **v** are already at the balanced value for the network and the proportion they send back. Only processors A and B are disproportionate. The results of this can be seen if we assume a state of balance on the network, where the workload of A should equal the workload of B. The equations reduce to:

$$A = A - (1/4\ A) + (1/3\ A) \qquad B = B - (1/3\ B) + (1/4\ B)$$
$$A = A + (1/12\ A) \qquad B = B - (1/12\ B)$$

It can be seen that if a balanced state is reached (A=B), the use of these diffusion proportions will immediately throw the system out of balance. Processor A will approach a value equal to 13/12 that of the network average, while processor B will approach a value equal to 11/12 of the network average. The workloads cannot become equivalent, and the difference in proportions causes this difficulty. It is obvious that the proportion of work leaving a processor should not exceed the proportion of work that enters it.

A method for ensuring equal proportions was implemented in the diffusion algorithm in this paper. Referring back to Figure 4, to reach a dynamic equilibrium processor B should not send the full ⅓ of its workload to processor A, because processor A may not be able to match it. In fact, since processor A can send at most 1/4 of its workload, and since 1/4 is below processor B's transmission maximum, the diffusion rule must allow processor B to reduce its proportion sent to match that of processor A.

In general terms, the proportion of work which a processor can transmit to a particular neighbor is guided by the following equation, where **y** is the number of immediate neighbors which the distribution processor has, **n** is the number of processor connections each neighbor has, and k is an arbitrary value that is greater than or equal to zero. In addition, MAX(y,n) returns y if y is greater than or equal to n, and returns n if n is greater than y.

$$1 / (MAX(y, n) + k)$$

This cellular automaton rule for diffusion (`diffuse`, `setconstant`) is implemented in AUTOMATA.C. This proportion results in a guaranteed equivalent exchange between any two processors. k may acquire any positive real value or zero, since increasing the denominator reduces the proportion of work being sent to neighbors, but never to an impossible amount. Varying this value affects the rate and manner of diffusion, and is studied in the next section.

5.0 Performance Evaluation and Comparison

In order to evaluate the effectiveness of varying the k value in the diffusion algorithm, some method of quantifying the quality and efficiency of a particular diffusion process is required. A good indicator of the load balance on a network is given by the standard deviation of the workloads on the processors [5]. The standard deviation can be calculated using the following equation, where **n** is the number of processors on the network, **a** is the average workload for processors on the network, and **x[i]** is any processor.

standard deviation = SQRT(SUM {i=1..n} of SQR(x[i]-a)) / n)

In Figure 2 presented earlier, the network standard deviation before any diffusion takes place is approximately 4.243, a large value. After one diffusion step, the standard deviation has been cut in half, to approximately 2.121. A second diffusion step lowers the standard deviation to approximately 1.061. It can be shown that the standard deviation is zero for any network that is completely balanced. This can be seen by inspection of the network on the right hand side of Figure 1 on page 16. The lower the standard deviation, the more distributed a network is considered to be. Routines for calculating the standard deviation on arbitrary networks (`calcstdev`, `getstdev`) are given in AUTOMATA.C.

Another way to evaluate the effectiveness of a particular algorithm is measurement of the amount of communication time necessary to achieve equilibrium. In the process of load balancing, tasks must be transmitted from one node in the network to another. This communication between nodes requires an amount of time proportional to the quantity of work that is sent. Referring back to Figure 2, it can be seen that the network starts out at zero communication time. Then 4.5 work units are transferred from the loaded processor to each of its neighbors. Although nine total work units are sent, they are sent in two parallel 4.5 unit packets, requiring only 4.5 communication time units. Routines for calculating the amount of communication time a generation in the automaton requires (`calcmaxdelta`, `getmaxdelta`) are included in AUTOMATA.C.

The final factor which can gauge the performance of a distribution algorithm in the cellular automaton model is the number of generations required to reach equilibrium Every generation requires the calculation of proportions and distributed work values, as well as computation to minimize communication flow, as was shown in Figure 3. The more generations that are required to reach equilibrium, the more time the system spends processing.

Because there are an infinite number of architectures which could be studied, the most common parallel computer structure was chosen for testing in this simulation. The simulation was run on a mesh architecture, consisting of a five by five grid of processors, with each processor connected to four other processors.

A concentrated load was placed in the center processor, and simulations were run until the system reached equilibrium. At this point, the standard deviation of the workloads on the processors is equal to zero, and the network is completely balanced. The value of k was varied between program runs. Source code for the simulation process is given in MESHTEST.C.

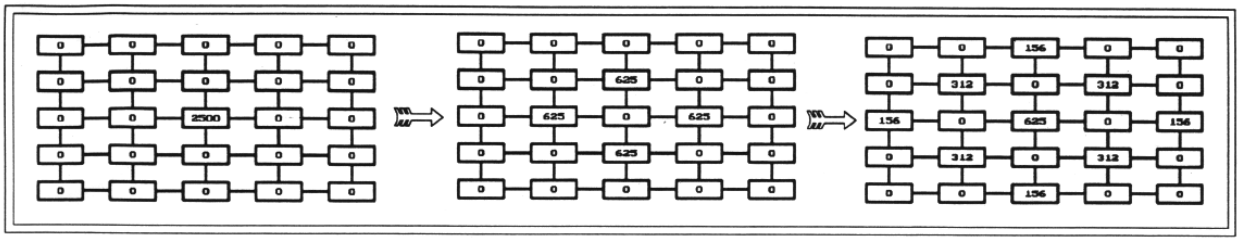

Figure 5: Two Generations on a 5x5 mesh using k = 0

Above are three successive generations of the mesh using a k value of zero, captured from the output of the mesh simulator. Since every processor has four connections and k is equal to zero, the proportion given to each neighbor is:

1 / (MAX(4, 4) + 0) = 1/4

This means that every processor in the mesh is giving away ¼ of its workload to each neighbor in every generation.

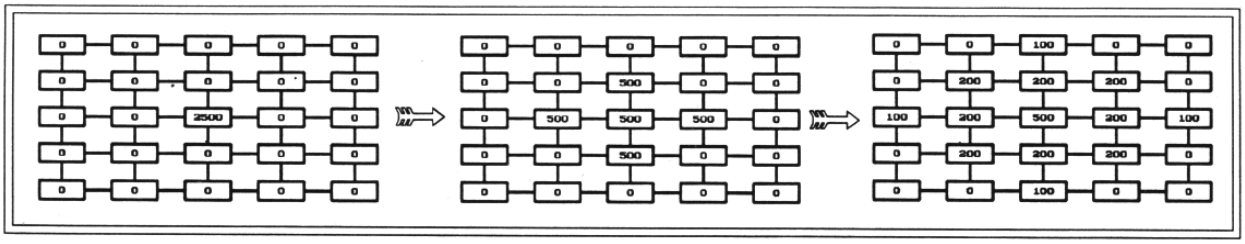

Figure 6: Two Generations on a 5x5 mesh using k = 1

Above is another example of diffusion on the mesh, in which a k value of 1 was used. Again, since every processor has four connections, and the k value is equal to one, the proportion given to each neighbor is:

1 / (MAX(4, 4) + 1) = 1/5

After each generation, the standard deviation and communication time were recorded. This simulation was performed for six different values of the constant, which was varied from zero to five. Some of the results can be seen in Figure 7.

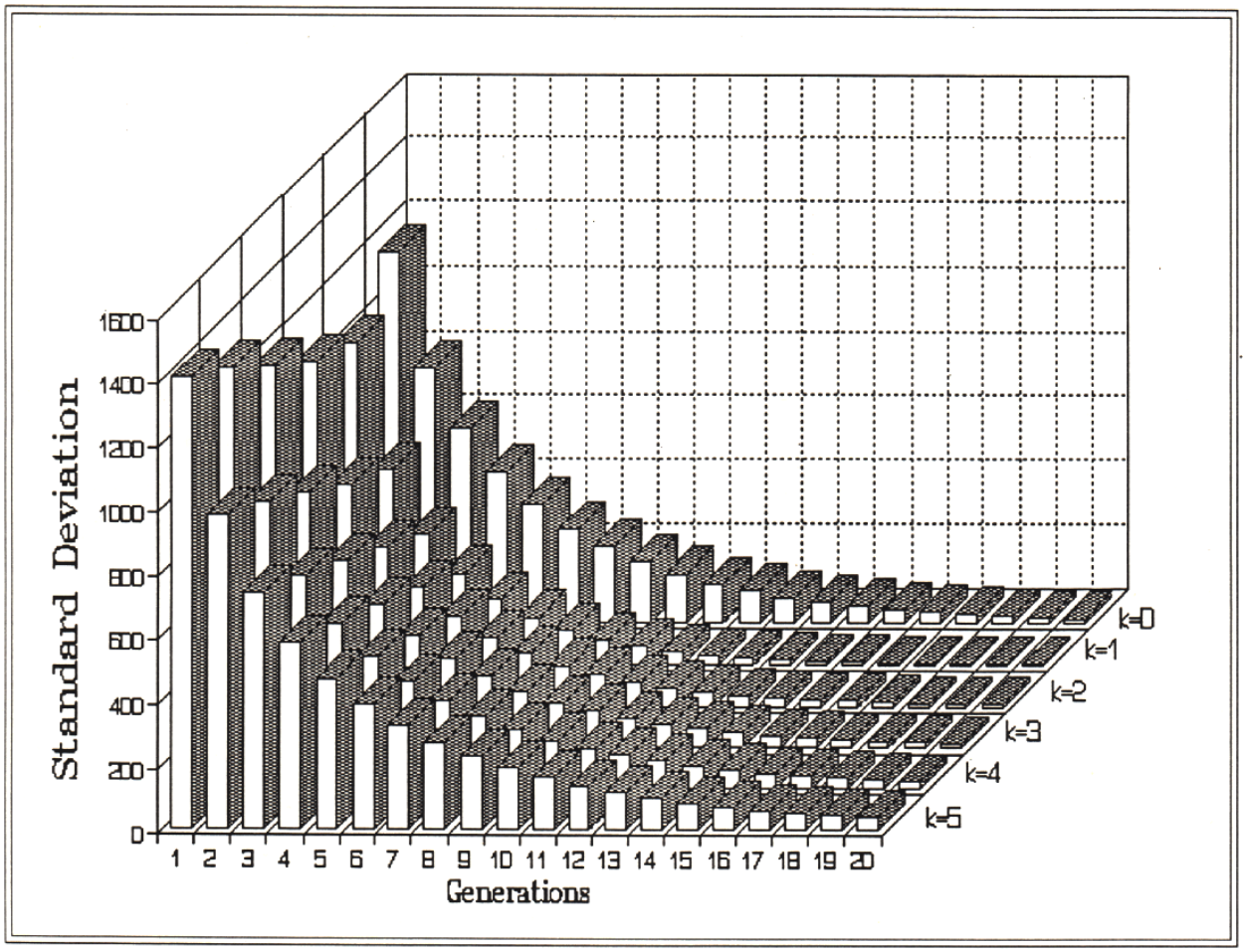

Figure 7: Graph of effects of varying k on automata generations to balance a 5x5 mesh.

This graph shows that equilibrium was reached in the fewest generations when the k value was equal to one. A k value of zero required many more generations than the k value of one. Values greater than one also required progressively more generations.

Figure 8 on the next page shows the standard deviation with respect to communication time for several different values of k. At a k value of zero, the most time was required, almost twice as much as was required when a value of one was used. The amount of communication time required decreased with increasing k, but the difference became less significant as k became larger than three.

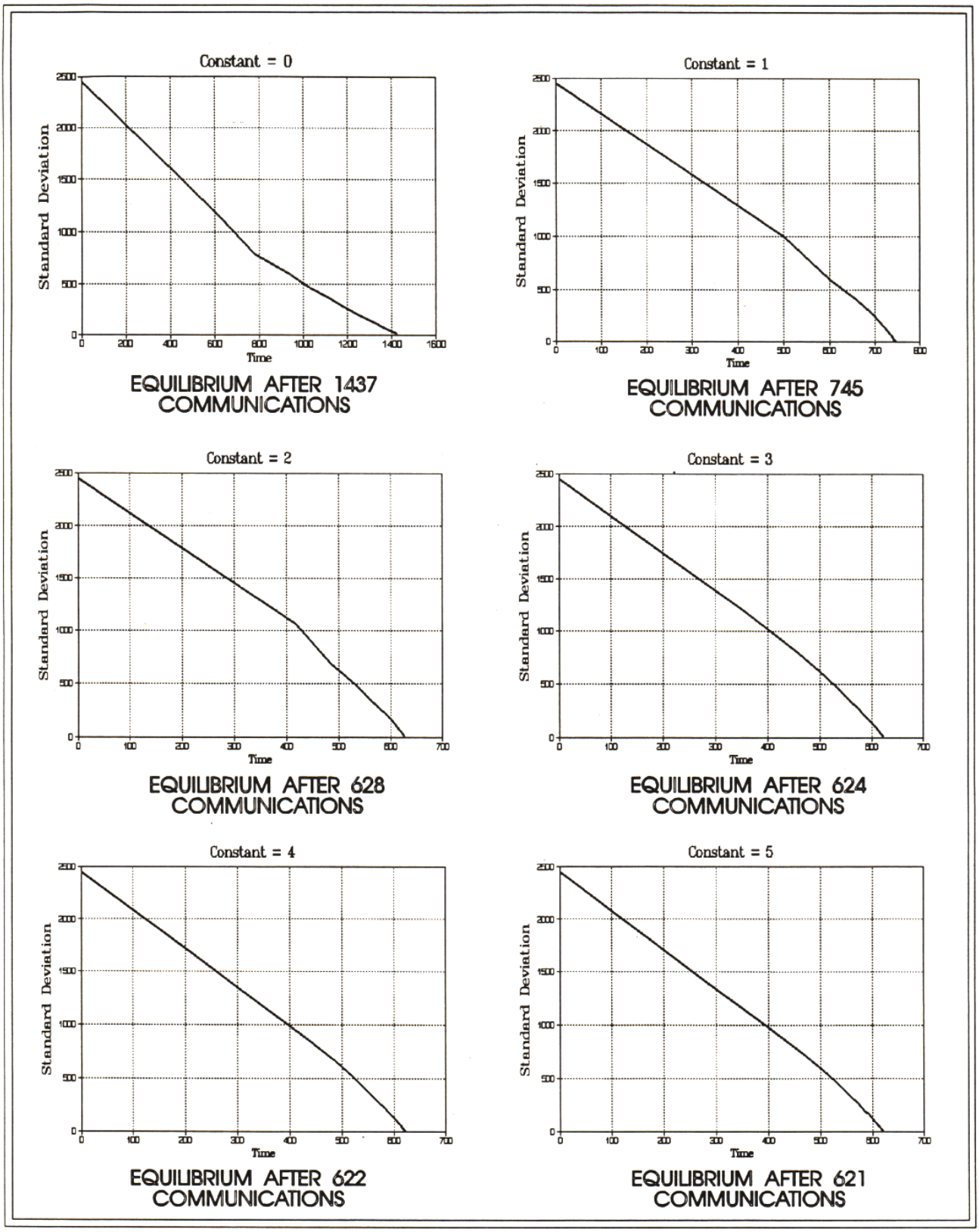

Figure 8: Communication time vs standard deviation for 2500 work units on a 5x5 mesh.

6.0 Conclusions and Future Research

In this project, it was observed that a cellular automaton approach to load balancing could be applied to parallel networks. The importance of a balance between the proportion of work entering a node and the proportion leaving that node was established, and a generalized diffusion rule to ensure equal rates was presented. This rule was of the form:

$$1 / (MAX(y, n) + k)$$

For the set of k values that were studied through simulation on a quadriconnected five by five mesh of processors, it was found that processing time (the number of generations required to reach equilibrium) could be minimized by using successively larger values of k. In a system where relative ratios between processing time and communication time are known, the optimal k value can be chosen. For those systems which have slower communications but faster processors, a larger k value should be used. For those systems which have slower processing but faster communications, a smaller k value should be used, but should not be less than one.

There is a possible explanation for the processing efficiency acquired when a k value of one is used, and the processing and communications inefficiency acquired when a k value of zero is used. The distribution proportion for a k value of zero is given by:

$$1 / (MAX(y, n) + 0)$$

In this expression, the processor is considering its neighbors as the only possible destinations for data flow. If a k value of one is used, the proportion is changed to:

$$1 / (MAX(y, n) + 1)$$

In this expression, the processor considers itself as a destination for data flow, in addition to its neighbors. A processor in a system using a k value of zero never keeps any of its workload between successive generations, as was seen in Figure 5. As a result, processors effectively eliminate themselves as participants in their own load balancing. By contrast, Figure 6 showed the same distribution performed with a k value of one. Thus, communications are reduced when the algorithm considers the donating processor to be a viable data flow destination during the load balancing.

Further research could allow for heterogeneous processing elements with variable speed, by weighting the processors by a constant factor. In other words, faster processors would give the illusion that they have less work to do than slower ones with the same number of jobs. This only requires a simple correction factor for processor speeds to the basic algorithm presented in this paper. More classes of network structures could be studied, and variations on the diffusion proportion expression for specific classes of networks could be suggested.

References

[1] Kequin Lib and Kam Hoi Cheng, "A Two Dimensional Buddy System for Dynamic Resource Allocation in a Partitionable Mesh Connected System," in Proceedings of Supercomputing '90, 1990

[2] Tai-Kuo Woo and Stanley Y. W. Su and Richard Newman-Wolfe, "Enhancing the Performance of a Dynamically Partitionable Bus Network by Using a Graph Coloring Algorithm," in Cooperation Proceedings ACM, February 20-22, 1990.

[3] Ahmed Louri, "A Symbolic Substitution Based Parallel Architecture and Algorithms for High-Speed Parallel Processing," in Cooperation Proceedings ACM, February 20-22, 1990.

[4] Charles M. Shub, "Native Code Process-Originated Migration in a Heterogeneous Environment," in Cooperation Proceedings ACM, February 20-22, 1990.

[5] Ishfaq Ahmad and Arif Ghafoor, "A Semi Distributed Task Allocation Strategy for Large Hypercube Supercomputers," in Proceedings of Supercomputing '90, 1990.

[6] Peter J Denning and Jeffrey P. Buzen, "Operational Analysis of Queueing Networks," in Measuring, Modelling and Evaluating Computer Systems, 1977.

[7] E. Gelenbe and G Pujolle, "A Diffusion Model for Multiple Class Queueing Networks," in Measuring, Modelling and Evaluating Computer Systems, 1977.

[8] Margolus, Normal and Toffoli, Tommaso. Cellular Automata Machines: A New Environment for Modeling. Massachusetts Institute of Technology Press: Cambridge, Massachusetts. 1987.

[9] Kotz, John C. and Purcell, Keith F. Chemistry & Chemical Reactivity. Saunders College Publishing: New York, NY. 1987.