Using Spanning-Trees for Balancing Dynamic Load on Multiprocessors

Rami G. Melhem, Kirk R. Pruhs, and Taieb F. Znati
Computer Science Department
University of Pittsburgh
Pittsburgh, PA 15260

Abstract

We consider the problem of load balancing to minimize the cost of dynamic computations, including the cost of migrations. We analyze the costs associated with diffusion based algorithms for several common architectures. We introduce the Ripple load balancing paradigm, which has several advantages over diffusion methods, including flexibility and faster convergence.

1 Introduction

We consider the problem of balancing dynamically changing loads on a processor graph assuming that execution proceeds in steps, with a local load balancing attempt after each step. We use $L_{i,t}$ to denote the load on node i at step t, and assume that the load is initially balanced. We make the standard assumption that there is a large number of tasks so that the load may be treated as a continuous variable [2]; This assumption can be removed by using the banking mechanism described in [6]. We will use c to denote the relative cost of a migration to computation; specifically, the cost of migrating a task between two neighboring processors is c times the cost of executing the task on a processor. The goal of a load balancing algorithm is thus to minimize the total computation time, taking into consideration the cost of migrations. Specifically, if the computation requires T steps, then the total computation cost is given by $\sum_{t=1}^{T}(\max\{L_{i,t}\} + c \max\{\lambda_{i,t}\})$, where $\lambda_{i,t}$ is the load migrated, at step t, from processor i to any neighboring processor. The inclusion of the migration cost in the analysis distinguishes this work from previous work on dynamic load balancing.

We propose the following criteria as a framework within which we can formally assess the effectiveness of load balancing algorithms. These criteria are defined with respect to a point of quiescence, ie. during load balancing no new tasks are created or consumed. The minimum conditions that a load balancing algorithm should meet are *stability*, the load eventually reaches a fixed distribution, and *levelness*, the load at all the processors is equal at the fixed distribution. The following measures determine the cost incurred to reach stability, which is our main concern in this paper:

1. *Time to Reach Stability* – The number $\hat{t}$ of load balancing steps necessary to reach stability.

2. *Migration Cost* – The time spent in migrating load until stability is reached. More formally, the migration cost is $\hat{M} = c\sum_{t=1}^{\hat{t}} \max\{\lambda_{i,t}\}$.

3. *Load Imbalance Cost* – The delay experienced in the computation steps due to load imbalance until leveling. Let $\hat{L}_t$ be the average load over all the processors at time t. The the load imbalance cost is formally defined as $\hat{G} = \sum_{t=1}^{\hat{t}} \max\{L_{i,t} - \hat{L}_t\}$.

While these criteria are defined from a point of quiescence, they can still be used to assess many aspects of the dynamic behavior of load balancing algorithms. This is because the dynamic behavior of the load balancing algorithms we consider can be described by superimposing the static behavior resulting from each load change.

2 Diffusion Algorithms

A standard class of local load balancing algorithms is diffusion (or sometimes called gradient) algorithms [2, 3, 7, 8]. Let L_t be a column vector with ith coordinate $L_{i,t}$. In a diffusion algorithm the loads are updated according to the formula $L_{t+1} = AL_t$, where A is an n by n nonnegative doubly stochastic matrix with $A_{i,j}$ representing the fraction of the load at P_j that is given to P_i. Cybenko [3] gives necessary and sufficient condition for a diffusion algorithm to be leveling. In the standard diffusion algorithm, on a degree d regular processor graph each, $A_{i,i} = A_{j,i} = 1/(d+1)$ for all P_j adjacent to P_i. If neighboring processors P_i and P_j want to exchange load at some time t, this can be implemented as follows. If $L_{i,t} > L_{j,t}$ then P_i migrates $A_{j,i}L_{i,t} - A_{i,j}L_{j,t}$ units of load to P_j.

As shown in Table 1, the standard diffusion algorithm may incur asymptotically nonoptimal costs in reaching leveling. The parameter d denotes the diameter of the graph; for a linear processor array $d = n$, for a 2-dimensional grid $d^2 = n$, and for the hypercube $2^d = n$. Note that the entries in this table assume that a load change of n occurs in one P_i, which can easily be seen to be the worst case for diffusion. These values were computed by noting the relationship between diffusion and random walks [5, 1]. In a random walk a particle moves from a node to an adjacent node at discrete time intervals. At each point it chooses each neighboring vertex with equal probability. If the particle starts at a vertex i the *convergence time* is defined to be the time until the probability that the particle is at vertex v is $O(1/n)$, for all vertices v. Then, one can see that the convergence time of the standard diffusion algorithm is bounded below by the convergence time of the random walk.

We now briefly explain how the costs for diffusion were computed. For the random walk, define $p_t(k)$ to be the probability that the particle is at a fixed vertex k units from the origin after t steps. One can show that on the line $p_t(0) = \Theta(1/\sqrt{t})$, for $t < n$. On the grid $p_t(0) = \Theta(1/t)$, for $t < n$. On the hypercube $p_t(0) = \Theta(2^{-d}(1 + e^{-2t/d})^d)$ [1]. In each architecture, for all t, $p_t(k)$ is a monotone decreasing function of k. These facts suffice to compute the time to reach stability and the load imbalance cost. Computing the migration cost of diffusion is not so easy because the maximum of the $\lambda_{i,t}$ may occur at a different P_i for each t.

3 Ripple Algorithms

We introduce a class of algorithms, called Ripple algorithms that reach leveling in time linear in the diameter of the processor graph. Ripple algorithms are based on the simple idea that if the load in the network is initially balanced, then, any load increase (or decrease) in one processor should be equally distributed among all the processors. We begin by describing the Ripple paradigm for a linear processor array. In particular, we present two algorithms; The first algorithm, *Tortoise*, minimizes migration cost, and the second algorithm, *Hare*, minimizes load imbalance cost.

A Ripple algorithm is constructed by superimposing n simple distributed load balancing algorithms, $\mathcal{A}_1, \ldots, \mathcal{A}_n$, where each $\mathcal{A}_i$ is an algorithm that is specifically designed to distribute the load forked at P_i uniformly to all other processors. Let $f_{i,0}$ be a change in load at P_i at time 0, and let $\mathcal{A}_i^j$ be the algorithm followed by the jth processor in $\mathcal{A}_i$. Note that $f_{i,0}$ could be negative. In the Ripple paradigm we distinguish between two types of load changes that a processor can experience. *New load* is load created or lost in a processor. *Passed load* is load that the processor gives to or receives from a neighbor. In $\mathcal{A}_i^i$, processor P_i keeps its fair share of the new load and passes the rest to P_{i-1} and P_{i+1}. A neighbor that receives a passed load, keeps its fair share of that load and continues passing the rest in the original direction.

When passing loads to neighboring processors we may not want to pass all the load in one step. Instead, we may want to schedule some of the load to be passed at future times. Different schedules will yield different Ripple algorithms. To implement this idea, each processor P_j maintains two tables $T_r^j(t)$ and $T_l^j(t)$ of reals, with $0 \leq t \leq m$, where m is the amount of time that it takes any $\mathcal{A}_i$ to balance the load. The variable t represents the current time. Each processor begins with $t = 0$ and increments t each balance step. The value of $T_r^j(k)$ and $T_l^j(k)$ represent the load that P_j will send to the right and left, respectively, at time k.

If P_i changes its load at some time t, other than 0, then the same tables can be used in a circular way. More specifically, the entry corresponding to t in the above tables will be $t \bmod m$. The algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_n$ may use the same tables to schedule the load to be passed.

Following is an informal description of $\mathcal{A}_i^j$ at time t for Tortoise.

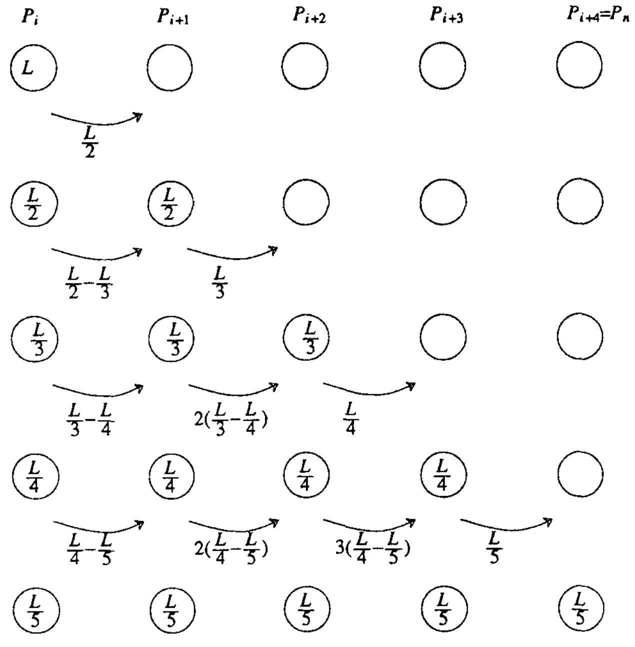
S1. If $i = j$ and $f_{i,t}$ is nonzero then:
Tortoise schedules the load to be passed to the left and the right. Specifically, it sets $T_l^i(t + a \bmod n) = T_l^i(t + a \bmod n) + f_{i,t}/n$, for $a = 0, \ldots, i-2$, and sets $T_r^i(t + a \bmod n) = T_r^i(t + a \bmod n) + f_{i,t}/n$, for $a = 0, \ldots, n-i-1$.

S2. P_j sends $T_l^j(t \bmod n)$ units of load to the left and $T_r^j(t \bmod n)$ units of load to the right. P_j then sets $T_l^j(t \bmod n)$ and $T_r^j(t \bmod n)$ to 0.

S3. If P_j receives ρ_r and ρ_l units of load from the right and left, respectively, it sets $T_l^j(t + 1 \bmod n) = T_l^j(t + 1 \bmod n) + \rho_r(j-1)/j$ and $T_r^j(t + 1 \bmod n) = T_r^j(t + 1 \bmod n) + \rho_l(n-j)/(n-j+1)$.

Note that in step S1, after scheduling the loads in T_l^i and T_r^i, P_i ends up keeping $f_{i,t}/n$ units of load, which is its fair share of $f_{i,t}$. Also note that in step S3, after scheduling the loads in T_l^j and T_r^j, P_j ends up keeping $\rho_r/j + \rho_l/(n-j+1)$ units of load, which is its fair share of ρ_r and ρ_l.

In a different algorithm, *Hare*, the goal is to achieve a load imbalance cost of $\Theta(n \log n)$, which may be shown to be optimal. In each $\mathcal{A}_i$, if the load change occurred at time 0, a schedule for Hare can be derived by imposing the condition that $L_{j,t}$ is at most $f_{i,0}/t$, if $|i-j| \leq t$ and zero otherwise. In Figure 1, we assume that L is the part of the load forked at P_i that should be equally distributed among $P_i, \ldots, P_n$.



P_i
P_{i+1}
P_{i+2}
P_{i+3}
$P_{i+4}=P_n$
L
$\frac{L}{2}$
$\frac{L}{2}-\frac{L}{3}$
$\frac{L}{3}$
$\frac{L}{3}-\frac{L}{4}$
$2(\frac{L}{3}-\frac{L}{4})$
$\frac{L}{4}$
$\frac{L}{4}-\frac{L}{5}$
$2(\frac{L}{4}-\frac{L}{5})$
$3(\frac{L}{4}-\frac{L}{5})$
$\frac{L}{5}$

Figure 1 – Minimizing the load imbalance in Hare

So far we have considered only linear interconnections. Given a general, point-to-point interconnection between n processors, $P_1, \ldots, P_n$, a Ripple algorithm, $\mathcal{A}_i$, may be used to distribute any load forked at P_i to all other processors uniformly. For this, a subset of the interconnection links should be chosen to form a spanning tree, S_i, rooted at P_i and connecting all the processors. Each processor, P_j should know the link, In_i^j, which connects it to its parent on S_i (if $j \neq i$) and the set of links, Out_i^j, which connect it to its children on S_i. P_j should also keep a scheduling table $T_{i,\ell}^j$ for each link ℓ in Out_i^j and should know the size of the subtree of S_i rooted at each of its children. Specifically, for each link, ℓ in Out_i^j connecting P_j to the root of a subtree of S_i, we denote the number of processors spanned by this subtree by $Size_{i,\ell}^j$, and we require that P_j knows $Size_{i,\ell}^j$.

When a load $f_{i,t}$ forks on P_i, P_i keeps $1/n$ of that load, and, for each ℓ in Out_i^i, it schedules $Size_{i,\ell}^i/n$ of $f_{i,t}$ in $T_{i,\ell}^i$ to be passed on link ℓ. When a processor, P_j receives a passed load, ρ, on In_i^j, it keeps $1/(u+1)$ of that load, where u is the sum of $Size_{i,\ell}^j$ for all ℓ in Out_i^j, and schedules $Size_{i,\ell}^j/(u+1)$ of ρ in $T_{i,\ell}^j$ to be passed to ℓ. As in the linear case, different algorithms may result from different scheduling of forked and passed loads.

Given that a different spanning tree S_i is associated with each forking processor, P_i, each processor P_j should keep a record of In_i^j, Out_i^j and $Size_{i,\ell}^j$ for each $i = 1, \ldots, n$. Moreover, the scheduling tables, $T_{i,\ell}^j$ at P_j may not generally be combined into one table, T_ℓ^j, because when P_j passes a load ρ on a link ℓ, it should pass with it the identity of S_i that is being used to propagate ρ. Fortunately, this book-keeping may be greatly simplified if the n spanning trees, S_i, $i = 1, \ldots, n$, satisfy the following condition:

$$\text{For any } j, \text{ if } In_i^j = In_k^j, \text{ then } Out_i^j = Out_k^j.$$

More descriptively, this condition states that, if two trees S_i and S_k are incident to P_j at the same incoming link, then the subtrees of S_i and S_k rooted at P_j are identical. Note that this implies that the maximum number of spanning trees passing by P_j is at most equal to the number of links connected to P_j. Moreover, the identity of the tree on which a load is being propagated need not be explicitly identified. When P_j receives a passed load ρ, the subtree on which ρ is to be propagated is uniquely identified by the link at which ρ was received. With this, only one scheduling table T_ℓ^j is needed for each link ℓ connected to P_j.

In order to illustrate this concept, we consider an $n = d^2$ processor grid, $P_{i,j}$, $i, j = 1, \ldots, d$. The above condition is satisfied if the spanning tree $S_{i,j}$ rooted at $P_{i,j}$ consists of all the links connecting processors in row i, and all the links connecting two processors in different rows. Note that the same undirected trees underlie $S_{i,j}$ and $S_{i,k}$, for $j, k = 1, \ldots, d$. For example, the trees $S_{2,3}$ and $S_{4,2}$ in the 16 node grid are shown in Figure 2.

In Figure 3, we consider $P_{3,3}$ and for each incoming link, we show the subtrees used to propagate a received load. Let $S_{i,j}(k,l)$ denote the subtree of $S_{i,j}$ rooted at $P_{k,l}$. Figure 3(a) shows $S_{3,4}(3,3)$. Figure 3(b) shows $S_{3,1}(3,3)$ $S_{3,2}(3,3)$, which are identical. Figure 3(c) shows $S_{i,j}(3,3)$, for $i = 1, 2$ and $j = 1, \ldots, 4$. Finally, Figure 3(d) shows $S_{4,j}(3,3)$, for $j = 1, \ldots, 4$.

Table 1 shows the asymptotic costs of two Ripple algorithms for the grid. The Hare (Tortoise) algorithm for the tree $S_{i,j}$ uses the linear Hare (Tortoise) algorithm to distribute the load evenly in the ith row, and also uses the linear Hare (Tortoise) algorithm to distribute the load in each column.

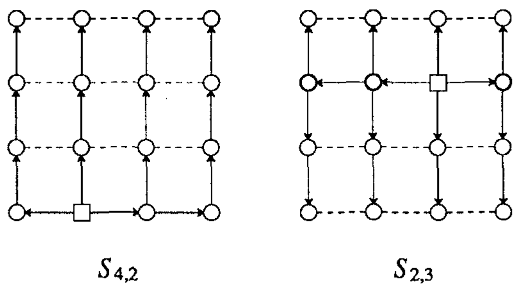


Figure 2 – Spanning trees $S_{2,3}$ and $S_{4,2}$

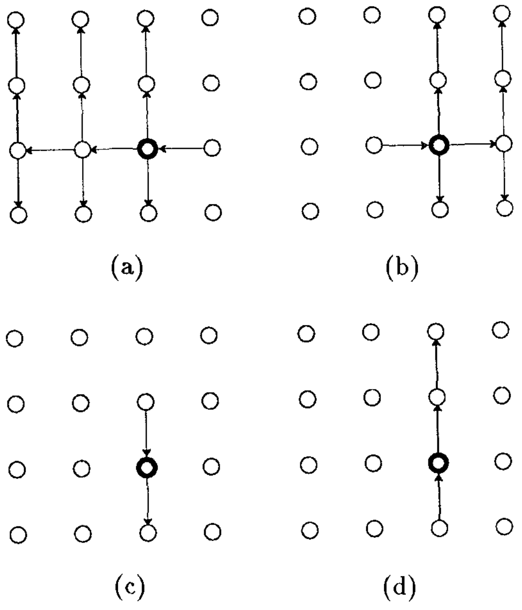


Figure 3 – Spanning Subtrees at $P_{3,3}$

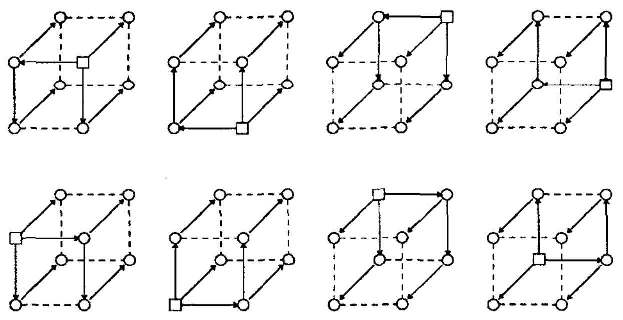


Figure 4 – The eight trees in a 3-D cube

For an $n = 2^d$ processor hypercube, the spanning binomial tree [4] rooted at a processor P_i may be used for S_i. This tree is constructed by connecting P_i to $N_k(P_i)$, $k = 1, \ldots, d$, where $N_k(P_i)$ denotes the neighbor of P_i across dimension k. Each $N_k(P_i)$ is then connected to $N_u(N_k(P_i))$ for $u = 1, \ldots, k-1$, and the process continues recursively. For example, the eight trees for a 3-dimensional cube are shown in Figure 4. Clearly, with these trees, a processor which receives a load ρ from the link across dimension k, keeps $1/2^{k-1}$ of ρ and passes $2^{u-1}/2^{k-1}$ of ρ to its neighbor across dimension u, for $u = 1, \ldots, k-1$.

Architecture Algorithm	Time to Stability	Imbalance Cost	Migration Cost
Line			
Diffusion	d^2	d^2	?
Tortoise	d	d^2	d
Hare	d	$d \log d$	$d \log d$
Grid			
Diffusion	d^2	$d^2 \log d$	?
Tortoise	d	d^3	d^2
Hare	d	d^2	$d^2 \log d$
Hypercube			
Diffusion	$d \log d$	2^d	?
Ripple	d	2^d	2^d

Table 1

Table 1 gives the asymptotic costs for different Ripple algorithms for various architectures. We briefly describe how these costs were computed. Clearly, the time to stability is $\Theta(d)$ in each case. Assume that $f_{i,0} = n$. First, we consider a linear processor array. The maximum load migrated by Hare in step t is n/t. Hence the migration cost is $\sum_{t=1}^{n} n/t = n\ H_n$, where H_n is the nth Harmonic number. Note that $H_n = \Theta(\log n)$. By the design of Hare, the maximum load at time t is n/t. Hence the load imbalance cost is $\sum_{t=1}^{n} n/t = \Theta(n \log n)$. The maximum load migrated by Tortoise in step t is 1, and hence the migration cost is $\Theta(n)$. In Tortoise, the maximum load at time t occurs at P_i and is $n-t$. Hence the load balance cost is $\sum_{t=1}^{n}(n-t) = \Theta(n^2)$.

Costs for the grid can easily be computed from the costs on linear processor array. For the hypercube the maximum load at time t is less than $2^d/2^t$, for $t < d$, and hence the load imbalance cost is $\Theta(2^d)$. Similarly, since the load transferred at time t is less than $n/2^t$, the load migration cost is $\Theta(2^d)$.

4 Conclusion

The Ripple technique introduced here has many advantages; its time to stability is $O(d)$, it can be viewed as both sender initiated and receiver initiated [7], and its scheduling mechanism allows it to be very flexible.

References

[1] Aldous, D., "Minimization Algorithms and Random Walk on the d-Cube," *Annals of Probability,* Vol. II, pp. 403–413, 1983.

[2] Bertsekas, D.P. , and Tsitsiklis, J.N., *Parallel and Distributed Computation: Numerical Methods,* Prentice Hall, 1989.

[3] Cybenko, G., "Dynamic Load Balancing for Distributed Memory Multiprocessors," *Journal of Parallel and Distributed Computing,* Vol. 7, pp. 279–301, 1989.

[4] Johnsson L. and Ho. C., "Spanning Graphs for Optimal Broadcasting and Personalized Communications in Hypercubes", *IEEE Transactions on Computers*, Vol. 38, pp. 1249–1268, 1989.

[5] Kemeny, J.G., and Snell, J.L. Snell, *Finite Markov Chains*, D. Van Nostrand Co., Princeton, N.J., 1960.

[6] Pruhs, K., Melhem, R., and Znati, T., "Dynamic Mapping of Adaptive Computations onto Linear Arrays", to appear in *Proceedings of Workshop on Unstructured Scientific Computation on Scalable Multiprocessors*, 1990.

[7] Willebeek-LeMair, M., and Reeves, A.P., "Dynamic Load Balancing Strategies for Highly Parallel Multicomputer Systems", Technical Report, EE-CEG-89-14, Dec. 1989.

[8] Willebeek-LeMair, M., and Reeves, A.P., International Conference on Parallel Processing, 1990.