An Improved Diffusion Algorithm for Dynamic Load Balancing

Y. F. Hu and R. J. Blake

Daresbury Laboratory, Daresbury,

Warrington WA4 4AD, United Kingdom

Tel. +44 (0)1925 603362, Fax +44 (0)1925 603634

e-mail: Y.F.Hu@dl.ac.uk

Abstract

Diffusion type algorithms [1, 3, 11] are some of the most popular algorithms for scheduling in dynamic load balancing. It is known however that this type of algorithm can suffer from slow convergence. In this paper the performance of the diffusion type algorithms is improved, while retaining the nearest neighbour communication requirement, through the use of Chebyshev polynomials. It is also proved that both the diffusion algorithm and the improved diffusion algorithm have an optimal property in terms of the amount of load migrated. Numerical results are given comparing the algorithm with the diffusion algorithm as well as a fast algorithm that requires global communication.

Keywords Dynamic load balancing; scheduling; diffusion algorithm; Chebyshev polynomials.

1 Introduction

An important issue in the efficient use of parallel computers is that of load balancing. For many applications, the load on each processor changes during the computation. Furthermore, on a none-dedicated network of computers, the performance of each computer can fluctuate. It is therefore very important to balance the computational load dynamically in an efficient manner.

To achieve the load balance, it is necessary to calculate the amount of load to be migrated from each processor to its neighbours. Algorithms for such calculations are called dynamic load balancing algorithms. Subsequently, it is also necessary to migrate the load based on this calculation, although this is not within the scope of the current paper.

Many dynamic load balancing algorithms exist, including diffusion type algorithms [1, 3, 11], the dimension exchange algorithm [3, 13, 14] and the multilevel algorithm [8]. Among them, the diffusion algorithms are frequently used because of their simplicity. The diffusion algorithms also require only local communication. However diffusion algorithms are known to suffer from slow convergence [1]. If applied to a ring topology, the number of iterations needed to reach a given tolerance is of $O(p^2)$, with p the number of processors in the ring. When p increases, the convergence deteriorates very quickly.

To overcome this slow convergence, the authors recently proposed an alternative algorithm [9]. The algorithm was demonstrated to be faster on parallel computers, and is now being used in the dynamic load balancing of irregular mesh based applications [12]. This algorithm also has the added advantage that the sum of the squares of the amount of load to be migrated is minimized, which helps to minimize the communication cost of the actual migration process. The disadvantage of this algorithm is that it requires global communication, in the form of global summations.

This paper attempts to improve the diffusion algorithm of [1, 3] and at the same time to retain its advantage of not requiring global communication. In Section 2, the diffusion algorithm is presented, and new variants of this algorithm proposed. An algorithm recently developed by the authors, which requires global communication, is introduced. An optimal property of the diffusion algorithm is proved. In Section 3 the improved diffusion algorithm is derived and its convergence analyzed. An optimal property is also established. Section 4 gives numerical results and Section 5 presents some conclusions.

2 The Diffusion Algorithm and A Global Algorithm

The dynamic load balancing algorithms considered here operate on the processor graphs. Depending on the application, the processor graph can be a graph that describes the topology of the parallel computer, or one that reflects the interconnection of the subdomains of a mesh that has been partitioned and distributed amongst processors.

Parallel finite element calculation with mesh refinement is one application where there is a need for dynamic load balancing, as the following simple example illustrates. Figure 1 (a) shows a mesh of an "A" shape partitioned into 8 subdomains. Figure 1 (b) shows the same mesh but with subdomain 1 refined. Due to this refinement, subdomain 1 on processor 1 has 25 mesh nodes, compared with 15 nodes for the other processors. This results in a load imbalance, that needs to be resolved by a dynamic load balancing step.

The corresponding processor graph for Figure 1 (b) is shown in Figure 2, where each vertex represents a subdomain (or a processor), and two vertices are linked by an edge if the corresponding subdomains share edges of the mesh. The load on each processor is shown in brackets.

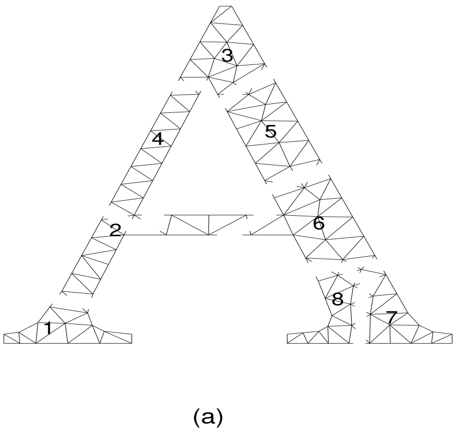

Figure 1: (a) A mesh of "A" shape partitioned into 8 subdomains; (b) the mesh refined at subdomain 1

2.1 Some Definitions

Let p be the number of processors. The processor graph is represented by a graph $(V,\ E)$, with $|V| = p$ vertices and $|E|$ edges. Vertices are numbered from 1 to p and edges are numbered from 1 to $|E|$. The graph is assumed to be connected. Denoting $i \leftrightarrow j$ if vertices i and j form an edge of the processor graph. Associated with each vertex (each processor) i is a scalar l_i representing the load on the processor. The average load per processor is

$$\bar{l} \ = \ \frac{\sum_{i=1}^{|V|} \ l_i}{|V|}. \tag{1}$$

Each edge is assumed to have a direction associated with it, say from the vertex with smaller index to the vertex with larger index. The former will be

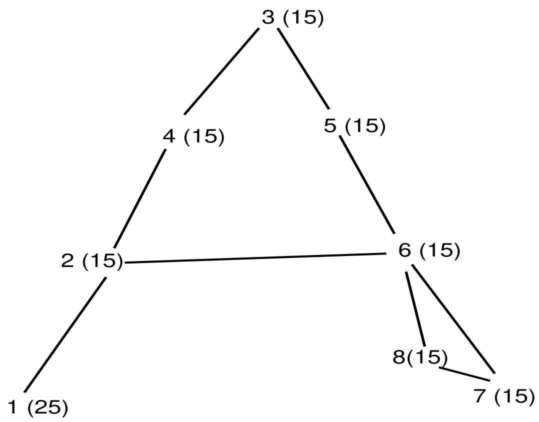

Figure 2: processor graph associated with the partitioned mesh in Figure 1 (b), and the load (in brackets) on each processor

called the *head* and the latter the *tail* of the edge. The amount of load to be transferred on the m-th edge (along the direction of the edge) is denoted as δ_m.

Denote

$$b = (l_1 - \bar{l}, l_2 - \bar{l}, \ldots, l_{|V|} - \bar{l})^T \tag{2}$$

the vector of load imbalance, and

$$x = (\delta_1, \delta_2, \ldots, \delta_{|E|})^T$$

the vector of the load to be transferred along the edges of the graph.

Definition The dynamic load balancing problem is that of finding a schedule that gives the amount of load δ_m to be transferred along any edge m, such that after the transfer, the load on each processor will be the same.

Consider the simple graph of Figure 3. The amount of load to be transferred along any edge is also called the "flow". The "flow", when satisfied, should make

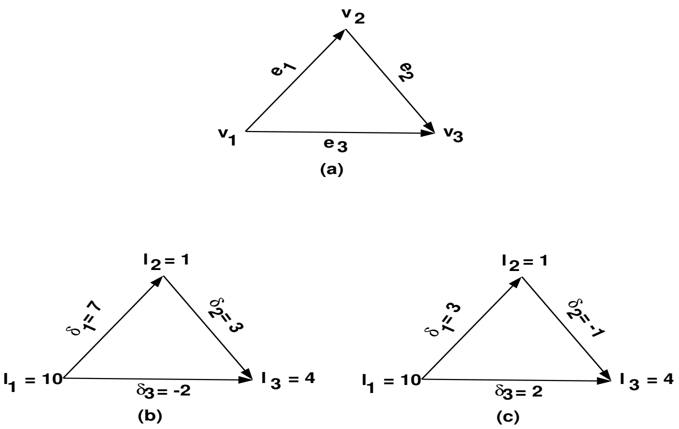

Figure 3: (a) a graph; (b) a load migration scheme; (c) another scheme

the load on each processor equal to the average load. That is,

$$\begin{aligned} \delta_1 + \delta_3 &= l_1 - \bar{l}, \\ -\delta_1 + \delta_2 &= l_2 - \bar{l}, \\ -\delta_2 - \delta_3 &= l_3 - \bar{l}. \end{aligned}$$

Or,

$$A\ x \ = \ b, \tag{3}$$

where

$$A = \begin{pmatrix} 1 & 0 & 1 \\ -1 & 1 & 0 \\ 0 & -1 & -1 \end{pmatrix}.$$

For a general graph, the matrix A is a matrix of dimension $|V| \times |E|$, termed the vertex-edge incident matrix [10], and defined as

$$(A)_{i,j} = \begin{cases} 1, & \text{if vertex } i \text{ is the head of edge } j, \\ -1, & \text{if vertex } i \text{ is the tail of edge } j, \\ 0, & \text{if vertex } i \text{ is neither the head nor the tail of edge } j. \end{cases} \tag{4}$$

2.2 The Diffusion Algorithm

The diffusion algorithm, as described in [1, 3], operates on the load itself. At each iteration of the algorithm, the new load $l_i^{(k+1)}$ of a vertex i is given by the combination of its original load $l_i^{(k)}$ and the load of its neighbouring vertices, namely

$$l_i^{(k+1)} \;=\; l_i^{(k)} \;-\; \sum_{i \leftrightarrow j} c_{ij}(l_i^{(k)} - l_j^{(k)}), \quad i,\ j \in V,\ k = 0, 1, 2, \ldots \tag{5}$$

Here c_{ij} are coefficients, usually less than one, representing the fact that only a fraction of the difference of load between processor i and its neighbours is sent/received. Initially the load for vertex $i \in V$ is $l_i^{(0)} = l_i$.

In matrix form, the above equation can be written as

$$l^{(k+1)} \;=\; (I \;-\; L)\; l^{(k)}, \quad k = 0, 1, 2, \ldots \tag{6}$$

where $l^{(k)} = (l_1^{(k)}, l_2^{(k)}, \ldots, l_{|V|}^{(k)})$ is the vector of load, the matrix

$$L = A\; W\; A^T$$

is called the weighted Laplacian matrix of the graph. It plays an important role in algebraic graph theory. Its explicit form can be found in equation (16). Matrix W, a diagonal matrix of the size $|E| \times |E|$, consists of the coefficients c_{ij}. Matrix A is the vertex-edge incident matrix (4).

For the choice of the coefficients, Boillat [1] suggested

$$c_{ij} = \frac{1}{\max\{deg(i), deg(j)\} + 1}, \quad i \leftrightarrow j, \quad i,\ j \in V. \tag{7}$$

Here $deg(i)$ stands for the degree of a vertex i, which corresponds to the number of edges attached to this vertex.

The amount of load transferred at iteration k from processor i to processor j is $c_{ij}(l_i^{(k)} - l_j^{(k)})$. Denote by $y^{(k)}$ a vector of dimension $|E|$, that consists of the load to be transferred along each of the edges at iteration k. The total amount of load to be transferred is then

$$x = \sum_{k=1}^{\infty} y^{(k)}. \tag{8}$$

Table 1 illustrates the diffusion algorithm applied to the processor graph of Figure 2.

2.3 A Unified Approach to Some Diffusion Algorithms

In this section we show how new variants of the diffusion algorithm can be derived, by relating the load balancing problem to the (continuous) diffusion equation for heat.

The 1-D diffusion equation is

$$\frac{\partial l}{\partial t} - D\,\frac{\partial^2 l}{\partial x^2} = 0.$$

Here l is the temperature, t is the time variable and x the space variable.

If the forward Euler scheme is used to discretize time and central differences are used to discretize space, then the equation becomes

$$\frac{l(x,t+\Delta t) - l(x,t)}{\Delta t} = -D\,\frac{l(x-\Delta x,t) - 2l(x,t) + l(x+\Delta x,t)}{(\Delta x)^2}, \tag{9}$$

or

$$l_i^{(k+1)} = l_i^{(k)} - \alpha\left\{ \left(l_i^{(k)} - l_{i-1}^{(k)}\right) + \left(l_i^{(k)} - l_{i+1}^{(k)}\right)\right\}.$$

This is just the diffusion algorithm (5), applied to a line. It is well known that this explicit Euler scheme is stable and convergent if and only if $\alpha < 0.5$.

If one applies the backward Euler discretization for time, one gets

$$\frac{l(x,t+\Delta t) - l(x,t)}{\Delta t} = -D\,\frac{l(x-\Delta x,t+\Delta t) - 2l(x,t+\Delta t) + l(x+\Delta x,t+\Delta t)}{(\Delta x)^2}. \tag{10}$$

or

$$l_i^{(k+1)} + \alpha\left\{ \left(l_i^{(k+1)} - l_{i-1}^{(k+1)}\right) + \left(l_i^{(k+1)} - l_{i+1}^{(k+1)}\right)\right\} = l_i^{(k)}.$$

For a general graph, in contrast to (6), the backward Euler scheme gives

$$(I + L)\, l^{(k+1)} = l^{(k)}, \quad k = 0, 1, 2, \ldots. \tag{11}$$

For partial differential equations (PDE's), the backward Euler scheme is known to be unconditionally stable. On graphs, it is easy to see that the above iterative process will converge for any $W > 0$ (recall that $L = AWA^T$), because the matrix on the left hand side is positive definite with all eigenvalues greater than one, except one eigenvalue that corresponds to the eigenvector of all ones. The explicit and implicit Euler methods (9) and (10) can be further combined to given the Crank-Nicholson scheme [7] for PDE's, namely

$$\frac{l(x, t + \Delta t) - l(x, t)}{\Delta t} = \nu R(1) + (1 - \nu) R(2), \quad \nu \in [0, 1] \tag{12}$$

where $R(1)$ and $R(2)$ are the right-hand-side of equations (9) and (10) respectively. Generalizing to graphs, the Crank-Nicholson scheme gives the following diffusion algorithm:

$$(I + \nu L)\, l^{(k+1)} = (I - (1 - \nu)\, L)\; l^{(k)}, \quad \nu \in [0, 1] \tag{13}$$

It is known [7] that the Crank-Nicholson scheme (12) is stable for $\nu \in [1/2, 1]$, the authors therefore conjecture that the diffusion algorithm (13) also converges for $\nu \in [1/2, 1]$.

A so called parabolic load balancing method was proposed by Heirich and Taylor [6] which was only applicable to mesh/torus like topologies. The method was in fact a special case of the implicit Euler method (11) applied to a graph with a uniform degree on all vertices, and with $W = \alpha I$. To preserve the nearest neighbour type communication, Heirich and Taylor used the Jacobi algorithm to solve the linear equation (11). Because the graphs they studied have a uniform degree of, say, s on all vertices, their method can be written simply as

$$l^{(k+\frac{j}{m})} = (1+\alpha s)^{-1}\left(l^{(k)} - \alpha(L - sI)l^{(k+\frac{j-1}{m})}\right), \quad j = 1, 2, \ldots, m;\ k = 1, 2, \ldots$$

Here m is the number of Jacobi iterations needed, chosen as a suitable integer. It is easy to verify, using the Gershgorin disc theorem [5], that the spectral radius of the iterative matrix $\alpha(L - sI)/(1+\alpha s)$ is less than $\alpha s/(1+\alpha s)$. Thus for a small time step α, the Jacobi iteration should converge very quickly. However a small time step would make the overall convergence to equilibrium slow, and the method would not have any particular advantage over the traditional (explicit Euler) diffusion algorithm.

A large time step would reduce the number of time steps to convergence considerably. In our numerical experiments we have experienced usually an order of magnitude reduction in the number of time steps compared with the explicit Euler method. However, a large time step α makes the spectral radius of the iterative matrix $\alpha(L-sI)/(1+\alpha s)$ close to unity, thus many Jacobi inner iterations are needed. Overall our numerical experiments and limited theoretical analysis on the Crank-Nicholson method (with implicit Euler method as a special case) have yet to find a suitable choice of α that would make the method substantially faster than the traditional (explicit Euler) diffusion algorithm, on general graphs other than meshes. This experience suggests that alternative approaches to improve the diffusion algorithm may be necessary. Section 3 gives one such attempt.

2.4 A Global Algorithm

An algorithm that is faster than diffusion type algorithms, but which requires global communication, was introduced in [9]. It is described here briefly as we shall be comparing it with the improved diffusion algorithm. The algorithm is motivated by the need to minimize the overall amount of load migration.

It was seen in Section 2.1 that the dynamic load balancing problem is equivalent to that of solving the linear system of equations (3). The matrix A is of size $|V| \times |E|$. So the linear system has $|V|$ equations and $|E|$ unknowns. As there are usually more edges than vertices in a graph (that is, $|E| > |V|$), there are usually more unknowns than equations and so the solutions to the problem are not unique.

As an example, for the graph in Figure 3 with the load as specified, the "flow" given by both Figure 3 (b) and Figure 3 (c) are valid. But schedule (c) is to be preferred as it involves less data movement.

Assuming that the weighted Euclidean norm of the data movement is used as a metric, then to minimize the data movement, the following problem needs to be solved

$$\begin{aligned} &\text{Minimize} \quad \frac{1}{2} x^T W^{-1} x, \\ &\text{subject to} \quad Ax = b. \end{aligned}$$

Here W is a diagonal weighting matrix of the size $|E| \times |E|$, representing the unit cost of communication between vertices (processors). Applying the necessary condition for the constrained optimization problem (see [4] and [9] for details) gives

$$x = WA^T d, \tag{14}$$

where d is the vector of Lagrange multipliers. Substituting (14) into $Ax = b$ gives

$$L\ d\ =\ b, \tag{15}$$

with $L = AWA^T$.

For the graph in Figure 3, assuming a unit weighting matrix of $W = I$, it is clear that

$$L = \begin{pmatrix} 2 & -1 & -1 \\ -1 & 2 & -1 \\ -1 & -1 & 2 \end{pmatrix}.$$

For general graph, if $W = I$, it can be confirmed [9] that the matrix L is in fact the Laplacian matrix of the graph, of dimension $|V| \times |V|$, defined as

$$(L)_{ij} = \begin{cases} -1, & \text{if } \ i \neq j, \ i \leftrightarrow j, \\ deg(i), & \text{if } \ i = j, \\ 0, & \text{otherwise.} \end{cases}$$

Here $deg(i)$ is the degree of vertex i in the graph.

If the weighting matrix W is not a unit matrix, then $L = AWA^T$ is the weighted Laplacian, of the form

$$(L)_{ij} = \begin{cases} -c_{ij}, & \text{if } \ i \neq j, \ i \leftrightarrow j, \\ \sum_{i \leftrightarrow k} c_{ik}, & \text{if } \ i = j, \\ 0, & \text{otherwise.} \end{cases} \tag{16}$$

Here c_{ij} is the entry of the weighting matrix on the edge between vertex i and j.

Thus the problem of finding an optimal load balancing schedule is transformed to that of solving the linear equation (15). Once the Lagrange multipliers are found, then the load transfer vector is $x = WA^T d$.

For any graph, each row m of the matrix A^T has only two nonzero entries, 1 and -1, corresponding to the head and tail vertices of the m-th edge. Therefore the amount of load to be transferred from processor i to processor j (assuming i is the head and j the tail), along the m-th edge (i, j), is simply

$$\delta_m = c_{ij}(d_i - d_j), \tag{17}$$

where d_i and d_j are the Lagrange multipliers associated with vertices i and j respectively. Because of the expression (17), the Lagrange multiplier d can be called the *potential.* To achieve load balance, the "flow" of load to be migrated along each edge of the graph is simply a weighted difference of the potentials at the two ends of the edge.

The linear system (15) can be solved using the parallel conjugate gradient algorithm [5], with a diagonal preconditioner. The parallel conjugate gradient algorithm involves global summations. This algorithm was found [9] to be in general much faster than the diffusion algorithm on a parallel Cray T3D computer.

As a simple example, considering the processor graph in Figure 2. The load for each processor is given in brackets. The average load is 16.25 and the largest load imbalance is $(25 - 16.25)/16.25 = 53.8\%$. The Laplacian system (15) (assuming that $W = I$) is now

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 3 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 2 & -1 & -1 & 0 & 0 & 0 \\ 0 & -1 & -1 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 2 & -1 & 0 & 0 \\ 0 & -1 & 0 & 0 & -1 & 4 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & -1 & -1 & 2 \end{pmatrix} \begin{pmatrix} d_1 \\ d_2 \\ d_3 \\ d_4 \\ d_5 \\ d_6 \\ d_7 \\ d_8 \end{pmatrix} = \begin{pmatrix} 25 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \\ 15 - 16.25 \end{pmatrix} = \begin{pmatrix} 8.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \\ -1.25 \end{pmatrix}.$$

The solution of this linear equation is

$$(d_1, \ldots, d_8) = (11.28, 2.53, -2.22, -0.47, -2.72, -1.97, -3.22, -3.22).$$

These potentials are illustrated in Figure 4 in brackets. The "flow" (the amount of load to be transferred) between two neighbouring processors is the difference

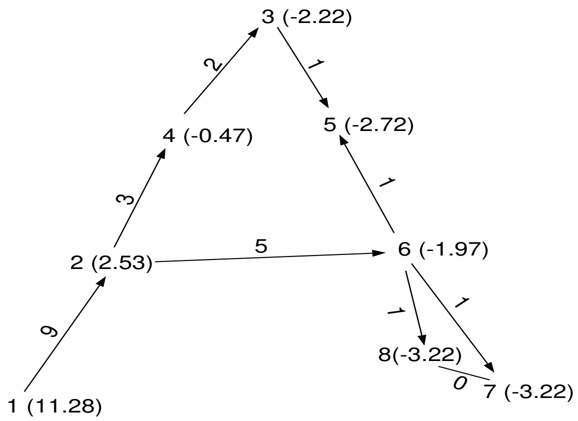

Figure 4: The potential and the amount to be migrated along each edge

between their potentials, and is shown along the edges in Figure 4. For example, processor 1 needs to send to processor 2 a load of $11.28 - 2.53 \approx 9$. After the "flow" is satisfied, the load for each processor will be (roughly) the same. For example vertex 1 will have a load of $25 - 9 = 16$, vertex 7 will also have a load of $15 + 1 + 0 = 16$.

Table 2 illustrates the algorithm applied to the processor graph of Figure 2. Here the load at iteration k is derived as $l^{(k)} = l^{(0)} - Ld^{(k)}$, where $d^{(k)}$ is the potential at iteration k.

Although this global algorithm is motivated by minimizing the Euclidean norm of the "flow", it came as a surprise from numerical experiments [9] that the Euclidean norm of the "flow" created by the diffusion algorithm was very close or equal to that of the global algorithm .

We can now prove that the diffusion algorithm also satisfies an optimal property.

2.5 An Optimal Property of the Diffusion Algorithm

The *induced graph* related to the diffusion algorithm (5) is defined as the original graph but with edges (i, j) removed if the coefficient $c_{ij} = 0$. It can be proved [3] that the diffusion algorithm will converge to the uniform distribution if the following assumption holds.

Assumption 1 *The diffusion algorithm (5) is assumed to satisfy the following*

- $c_{ij} \geq 0, \quad \text{for } i \leftrightarrow j;$
- $\sum_{j:\ i \leftrightarrow j} c_{ij} < 1, \quad \text{for } i \in V;$
- $c_{ij} = c_{ji};$
- *the induced graph is connected.*

Furthermore, the matrix L is a symmetric positive semi-definite matrix with a row-sum of zero, and its eigenvalues satisfy the following [3].

Theorem 1 *Under Assumption 1, the matrix L has eigenvalues*

$$2 > \lambda_{|V|} \geq \lambda_{|V|-1} \geq \ldots \geq \lambda_2 > \lambda_1 = 0 \tag{18}$$

and that the vector $z = (1, 1, \ldots, 1)^T$ is within scaling the only eigenvector corresponds to the eigenvalue of 0.

The "flow" given by the diffusion algorithm upon convergence, is the total accumulated load transfer. As pointed out before, the diffusion algorithm will converge to the uniform load under Assumption 1. It is proved in the following that the sum of the load transfer at each iteration also converges.

At iteration k of the diffusion algorithm, the amount of load transferred, along the m-th edge (i, j), equals the load difference between the two vertices, scaled by the edge weight. That is,

$$\delta_m^{(k)} = c_{ij} * (l_i^{(k)} - l_j^{(k)}).$$

In matrix form this is

$$y^{(k)} \;=\; W\; A^T\; l^{(k)},$$

where

$$y^{(k)} = (\delta_1^{(k)}, \delta_2^{(k)}, \ldots, \delta_{|E|}^{(k)})^T$$

is the vector of "flow" along the edges at iteration k. The sum of the load transferred after the first k iterations on each of the edges is given by the vector

$$\begin{aligned} \sum_{i=0}^{k} y^{(i)} &= W\; A^T \sum_{i=0}^{k} l^{(i)} \\ &= W\; A^T \left(\sum_{i=0}^{k} (I - L)^i \right) l^{(0)}. \end{aligned}$$

Theorem 2 *For the diffusion algorithm (5), under Assumption 1, the sum of the load transferred converges, that is, the vector* $\sum_{i=0}^{k} y^{(i)}$ *converges as* $k \to \infty$.

Proof Since L is a symmetric matrix, the vector of initial load $l^{(0)}$ can be expressed as the linear combination of $|V|$ mutually orthogonal eigenvectors of the matrix L,

$$l^{(0)} \;=\; \sum_{i=1}^{|V|} a_i\; u_i,$$

where u_i is the eigenvector corresponds to the i-th eigenvalue λ_i of the matrix L. Because of Theorem 1, it can be assumed that $u_1 = z = (1, 1, \ldots, 1)^T$ and $a_1 = z^T l^{(0)}/|V| = \bar{l}$.

Thus

$$\begin{aligned} \sum_{i=0}^{k} y^{(i)} &= W\; A^T \left(\sum_{i=0}^{k} (I - L)^i \right) l^{(0)} \\ &= W\; A^T \left(\sum_{j=1}^{|V|} \left(\sum_{i=0}^{k} (1 - \lambda_j)^i \right) a_j u_j \right). \end{aligned}$$

Because $A^T u_1 = A^T z = 0$, and $\sum_{i=0}^{k}(1-\lambda_j)^i \to 1/\lambda_j$ for $j \geq 2$ (notice that from Theorem 1, $|1-\lambda_i| < 1$ for $i \geq 2$), thus $\sum_{i=0}^{k} \; y^{(i)}$ converges to

$$W \; A^T \sum_{j=2}^{|V|} \frac{a_j}{\lambda_j} u_j.$$

$\square$

Denote d as the vector

$$d = \sum_{j=2}^{|V|} \frac{a_j}{\lambda_j} u_j.$$

From the proof of Theorem 2 the total accumulated amount of load transferred, in other words the "flow" calculated by the diffusion algorithm, is given by the vector

$$x \; = \sum_{i=1}^{\infty} y^{(k)} \; = \; W \; A^T \; d. \tag{19}$$

Having proved that the accumulated load migration given by the diffusion algorithm converges, one can now prove that this "flow" is optimal in the following sense:

Theorem 3 *Under Assumption 1, the total amount of load migration x (the "flow") generated by the diffusion algorithm (5) is the solution of the following minimization problem*

$$\begin{aligned} &\text{Minimize} \;\; \tfrac{1}{2} x^T W^{-1} x, \\ &\text{subject to} \quad Ax = b. \end{aligned} \tag{20}$$

Here W is the diagonal matrix of edge weights, these weights are assumed to be positive so that W^{-1} exists. The matrix A is the vertex-edge incident matrix defined in (4) and the vector b is the vector of load imbalance (2).

Proof Because the load migration scheme generated by the diffusion algorithm satisfies (19), it can be confirmed that $Ax = b$. This is because

$$A \; x \quad = \quad A \, W \; A^T \; d = L \; d$$

$$
\begin{aligned}
&= L\sum_{i=2}^{|V|}\frac{a_i}{\lambda_i}u_i \\
&= \sum_{i=2}^{|V|}a_iu_i \\
&= l^{(0)} - \bar{l}z = b.
\end{aligned}
$$

Now if x is not the minimum of (20), then there exists a vector $\Delta x \neq 0$ such that

$$A(x+\Delta x) = b$$

and

$$(x+\Delta x)^T W^{-1}(x+\Delta x) \; < \; x^T W^{-1} x.$$

Therefore

$$A\Delta x \; = \; 0$$

and

$$2x^T W^{-1}\Delta x + (\Delta x)^T W^{-1}\Delta x \; < \; 0. \tag{21}$$

Because of (19), the first term in (21) becomes

$$2x^T W^{-1}\Delta x \; = \; 2(WA^Td)^T W^{-1}\Delta x \; = \; 2d^T A\Delta x = 0,$$

so (21) gives

$$(\Delta x)^T W^{-1}\Delta x \; < \; 0. \tag{22}$$

This inequality is not possible since W is a diagonal matrix consisting of all positive elements and $\Delta x \neq 0$. Thus x has to be the solution of the minimization problem (20). □

This theorem implies that the "flow" generated by the diffusion algorithm is exactly the same as that generated by the global algorithm, provided that the same weighting matrix W is used for both algorithms. The fact that the diffusion algorithm actually minimizes a weighted Euclidean norm of the "flow" is perhaps not surprising after all, since the diffusion algorithm mimics the process of diffusion in nature, and nature always follows the principle of minimum energy!

The optimal property of the diffusion algorithm is not an endorsement of the algorithm. The diffusion algorithm suffers from poor convergence on graphs that are not rich in connectivity [1, 3, 9]. However, in developing newer and faster algorithms, it is sensible, from the point of view of reducing the communication cost, to make sure that they also have this optimal property.

3 Improving the Diffusion Algorithm

The diffusion algorithm, as described by equation (6), is in the form of a classical stationary iterative algorithm. As such it is susceptible to slow convergence when the spectral radius of the iterative matrix is close to one.

Let $\bar{l}$ be the average load, as defined in (1). Let z be the vector of all ones. Then the load imbalance $e^{(k+1)}$ at iteration $k+1$ is $e^{(k+1)} = l^{(k+1)} - \bar{l}z$. Let $L = AWA^T$ be the weighted Laplacian matrix (16), from (6) one has

$$l^{(k+1)} = (I - L)\, l^{(k)}.$$

Subtract both side by $\bar{l}z$, because $Lz = 0$,

$$e^{(k+1)} = (I - L)\, e^{(k)} = (I - L)^{k+1}\, e^{(0)}.$$

3.1 Motivation and Preliminaries

For the rest of this paper it is assumed that Assumption 1 holds. The motivation of the improved diffusion algorithm is as follows. Consider a two-step diffusion algorithm, $e^{(k+1)} = (I - L)^2 e^{(k-1)} = q_2(L)e^{(k-1)}$, where $q_2(\lambda) = (1 - \lambda)^2 = 1 - 2\lambda + \lambda^2$ is a polynomial of degree 2. However this polynomial may not be the one which makes the norm of the load imbalance $||q_2(L)e^{(k-1)}||$ small. It is possible that there is a polynomial of degree 2, $p_2(\lambda) = 1 + a\lambda + b\lambda^2$, such that $||p_2(L)e^{(k-1)}||$ is smaller. A 2-step diffusion algorithm based on the polynomial p_2 would therefore converge faster.

In general, for the diffusion algorithm, the load imbalance satisfies

$$e^{(k)} = q_k(L)e^{(0)}$$

where $q_k(\lambda) = (1-\lambda)^k$ is a polynomial of degree k. Here again this particular polynomial q_k may not be close to the best one. To improve the diffusion algorithm, it is necessary therefore to choose a polynomial p_k such that $||p_k(L)e^{(0)}||$ is minimized, or at least approximately minimized. The improved diffusion algorithm is then $l^{(k)} = p_k(L)l^{(0)}$. In order that the summation of the load at each iteration is conserved, it is necessary that $z^T l^{(k)} = z^T l^{(0)}$, with z the vector of all ones. Because z is an eigenvector of L of zero eigenvalue, this requires that $z^T l^{(k)} = z^T p_k(L) l^{(0)} = p_k(0) z^T l^{(0)}$, or $p_k(0) = 1$.

To facilitate the analysis, let u_i be the normalized eigenvector corresponding to eigenvalue λ_i of the weighted Laplacian matrix. Because L is symmetric it can be assumed that the u_i are orthogonal to each other and that the initial imbalance $e^{(0)}$ can be expressed as the sum of this orthogonal set

$$e^{(0)} = \sum_{i=1}^{|V|} a_i u_i.$$

Due to Theorem 1, one can assume that $u_1 = z = (1, 1, \ldots, 1)^T$. Notice that the sum of load imbalances over all vertices should be zero,

$$z^T e^{(0)} = u_1^T e^{(0)} = a_1 ||u_1||^2 = 0,$$

this means that $a_1 = 0$. Therefore

$$e^{(k)} = p_k(L)e^{(0)} = \sum_{i=2}^{|V|} a_i p_k(\lambda_i) u_i,$$

and the Euclidean norm of the error is

$$||e^{(k)}||^2 = \sum_{i=2}^{|V|} \left(p_k(\lambda_i)\right)^2 (a_i)^2 \leq \left(\max_{i=2}^{|V|} |p_k(\lambda_i)|^2 \right) ||e^{(0)}||^2. \tag{23}$$

Therefore to give the load imbalance the minimum possible upper-bound, the polynomial p_k should be chosen such that

$$\max_{i=2}^{|V|} |p_k(\lambda_i)| \tag{24}$$

is minimized, in addition to $p_k(0) = 1$.

Unfortunately, the polynomial which minimizes the term (24) is not readily solvable without explicit knowledge of all the positive eigenvalues λ_i, $i = 2, 3, \ldots, |V|$. However, as an approximation, taking into account the fact that

$$\max_{i=2}^{|V|} |p_k(\lambda_i)| \;\leq\; \max_{\lambda_2 \leq \lambda \leq \lambda_{|V|}} |p_k(\lambda)|,$$

one can seek a polynomial of degree k, such that $p_k(0) = 1$ and the maximum error over the interval $[\lambda_2, \lambda_{|V|}]$,

$$\max_{\lambda_2 \leq \lambda \leq \lambda_{|V|}} |p_k(\lambda)|, \tag{25}$$

is minimized. By replacing (24) with (25), one has made an approximation. In return, this simplified min-max problem over the continuous interval has a known solution, expressed in terms of the Chebyshev polynomials [15].

The Chebyshev polynomial $T_k(\lambda)$ is a polynomial of degree k such that

$$T_{k+1}(\lambda) = 2\lambda T_k(\lambda) - T_{k-1}(\lambda) \tag{26}$$

and

$$T_0(\lambda) = 1, \quad T_1(\lambda) = \lambda$$

Over the interval $[-1, 1]$, the polynomial $\frac{1}{2^{k-1}} T_k(\lambda)$ is the polynomial closest to zero, among all polynomials of degree k that have a coefficient of one for the highest order term. The Chebyshev polynomials can also be written as

$$T_k(\lambda) = \cos(k \arccos(\lambda)), \quad |\lambda| \leq 1, \tag{27}$$

or

$$T_k(\lambda) = \frac{1}{2}\left[(\lambda - \sqrt{\lambda^2 - 1})^k + (\lambda + \sqrt{\lambda^2 - 1})^k\right], \qquad |\lambda| \geq 1. \tag{28}$$

The polynomial of degree k, which satisfies $p_k(0) = 1$ and minimizes the maximum error over the interval $[\lambda_2, \lambda_{|V|}]$, can be expressed in terms of the Chebyshev polynomials [15] as

$$p_k(\lambda) = \frac{T_k(\xi(\lambda))}{T_k(\xi(0))}, \tag{29}$$

where

$$\xi(\lambda) = \frac{2\lambda - \lambda_2 - \lambda_{|V|}}{\lambda_{|V|} - \lambda_2}$$

is a linear transformation from $[\lambda_2, \lambda_{|V|}]$ to $[-1, 1]$.

3.2 Calculation of the Loads

Having found (approximately) the best polynomial, it is necessary to find a recursive procedure for the iteration

$$l^{(k+1)} = p_{k+1}(L) l^{(0)}.$$

Denote $C_k = T_k(\xi(0))$ and $Y = \xi(L) = (2L - \lambda_2 - \lambda_{|V|})/(\lambda_{|V|} - \lambda_2)$, then by (26),

$$\begin{aligned} l^{(k+1)} &= \frac{T_{k+1}(Y)}{C_{k+1}} l^{(0)} = \frac{2Y T_k(Y) - T_{k-1}(Y)}{C_{k+1}} l^{(0)} \\ &= \left(\frac{4L}{\lambda_{|V|} - \lambda_2} + 2\xi(0)\right) \frac{C_k}{C_{k+1}} l^{(k)} - \frac{C_{k-1}}{C_{k+1}} l^{(k-1)}. \end{aligned}$$

Because of (26),

$$C_{k+1} = 2\xi(0) C_k - C_{k-1}, \tag{30}$$

let $\alpha_{k+1} = 2\xi(0) C_k / C_{k+1}$, then

$$l^{(k+1)} = \alpha_{k+1} \left(1 + \frac{4L}{2\xi(0)(\lambda_{|V|} - \lambda_2)}\right) l^{(k)} + (1 - \alpha_{k+1}) l^{(k-1)}.$$

Substituting $\xi(0) = -(\lambda_2 + \lambda_{|V|})/(\lambda_{|V|} - \lambda_2)$ into the above equation gives

$$l^{(k+1)} = \alpha_{k+1}\left(I - \frac{L}{\beta}\right) l^{(k)} + (1 - \alpha_{k+1}) l^{(k-1)}, \quad k = 1, 2, \ldots \tag{31}$$

where $\beta = (\lambda_2 + \lambda_{|V|})/2$. For the first iteration,

$$l^{(1)} = \frac{T_1(\xi(L))}{C_1} l^{(0)} = (I - \frac{L}{\beta}) l^{(0)}. \tag{32}$$

The coefficient α_{k+1} in (31) can also be calculated recursively. Dividing both sides of (30) by $2\xi(0)C_k$ gives

$$\frac{C_{k+1}}{2\xi(0)C_k} = 1 - \frac{C_{k-1}}{2\xi(0)C_k},$$

or

$$\alpha_{k+1} = \frac{1}{1 - \alpha_k g}, \quad k = 1, 2, \ldots$$

where $g = 1/(2\xi(0))^2$. The iteration starts with

$$\alpha_1 = 2\xi(0)C_0/C_1 = 2\xi(0)/\xi(0) = 2.$$

3.3 Calculation of the Potentials

Having worked out the load at each vertex during each iteration, it is necessary to know the amount of load transferred during each iteration. For this purpose the idea of the potential, described in equation (17) of Section 2.4, is utilized. Define the potential at iteration k as $d^{(k)}$, a vector of size $|V|$. The total amount of load transfer up to iteration k is

$$x^{(k)} = WA^T d^{(k)} \tag{33}$$

The current load on each vertex is therefore the initial load minus the load transferred, that is,

$$l^{(k)} = l^{(0)} - AWA^T d^{(k)} = l^{(0)} - Ld^{(k)}. \tag{34}$$

At this point it is not certain that such a potential vector $d^{(k)}$ exists for the iterative process defined by (32) and (31). The existence of the potential vectors shall be proven in the following by induction.

For $k = 0$, equation (34) is clearly valid with

$$d^{(0)} = 0.$$

For $k = 1$, because of (32), equation (34) is valid with

$$d^{(1)} = \frac{1}{\beta} l^{(0)}.$$

Assume that equation (34) is valid for up to iteration k, then from (31),

$$\begin{aligned}
l^{(k+1)} &= \alpha_{k+1}\left(I - \frac{L}{\beta}\right) l^{(k)} + (1-\alpha_{k+1}) l^{(k-1)} \\
&= \frac{-\alpha_{k+1}}{\beta} L l^{(k)} + \alpha_{k+1}\left(l^{(0)} - L d^{(k)}\right) + (1-\alpha_{k+1})\left(l^{(0)} - L d^{(k-1)}\right) \\
&= l^{(0)} - L\left[\alpha_{k+1}\left(d^{(k)} + \frac{1}{\beta} l^{(k)}\right) + (1-\alpha_{k+1}) d^{(k-1)}\right].
\end{aligned}$$

It follows that (34) is valid at iteration $k+1$ with

$$d^{(k+1)} = \alpha_{k+1}\left(d^{(k)} + \frac{1}{\beta} l^{(k)}\right) + (1-\alpha_{k+1}) d^{(k-1)}, \quad k = 1, 2, \ldots. \tag{35}$$

3.4 Calculation of the Load Transfer

The transfer of load can be executed in cumulative fashion, when the algorithm has converged, using formula (33). Alternativly it can be carried out at each iteration. Denote by $y^{(k+1)} \in R^{|V|}$ the load transfer at iteration $k+1$. This load transfer is the difference of total load transferred up to iteration $k+1$, and that up to iteration k. That is,

$$y^{(k+1)} = x^{(k+1)} - x^{(k)} = WA^T(d^{(k+1)} - d^{(k)}).$$

Substituting (35) into the above equation gives

$$\begin{aligned}
y^{(k+1)} &= WA^T\left[\alpha_{k+1}\left(d^{(k)} + \frac{1}{\beta}l^{(k)}\right) + (1-\alpha_{k+1})d^{(k-1)} - d^{(k)}\right] \\
&= WA^T\left[(\alpha_{k+1}-1)(d^{(k)} - d^{(k-1)}) + \frac{\alpha_{k+1}}{\beta}l^{(k)}\right] \\
&= (\alpha_{k+1}-1)y^{(k)} + \frac{\alpha_{k+1}}{\beta}WA^Tl^{(k)}, \quad k = 1, 2, \ldots
\end{aligned}$$

For the first iteration

$$y^{(1)} = WA^T(d^{(1)} - d^{(0)}) = \frac{1}{\beta}WA^Tl^{(0)}$$

3.5 The Improved Diffusion Algorithm

The improved diffusion algorithm can now be stated in full as follows. The algorithm requires the knowledge of the largest and smallest positive eigenvalues of the weighted Laplacian matrix. The following notation is adopted:

- $\lambda_2,\ \lambda_{|V|}$: minimum and maximum positive eigenvalues of the weighted Laplacian matrix L;
- l_i: current load on processor i;
- d_i: current potential on processor i;
- y_{ij}: amount of load to be transferred from processor i to the neighbour processor j at this iteration;
- c_{ij}: the coefficients that satisfied Assumption 1.

Algorithm CHEBY *The improved diffusion algorithm*

- *1. Initialization:* $\beta = (\lambda_2 + \lambda_{|V|})/2,\ \gamma = (\lambda_{|V|} - \lambda_2)/2,\ g = \gamma^2/(4\beta^2)$ *and* $d_i^{old} = 0,\ i \in V.$

- *2. For each vertex $i \in V$, do $k = 1, 2, \ldots$ while the stopping criterion not satisfied*
  - *if ($k = 1$) then*

    a). $\alpha = 2$,

    b). $d_i \leftarrow l_i/\beta$,

    c). $y_{ij} = c_{ij}(l_i - l_j)/\beta, \quad i \leftrightarrow j$,

    d). $l_i \leftarrow l_i - \sum_{i \leftrightarrow j} y_{ij}$.
  - *else*

    e). $\alpha \leftarrow 1/(1 - \alpha g)$,

    f). $d_i^{temp} = d_i$,

    g). $d_i \leftarrow \alpha(d_i + l_i/\beta) + (1 - \alpha)d_i^{old}$,

    h). $d_i^{old} = d^{temp}$,

    i). $y_{ij} \leftarrow (\alpha - 1)y_{ij} + \alpha c_{ij}(l_i - l_j)/\beta, \quad i \leftrightarrow j$,

    j). $l_i \leftarrow l_i - \sum_{i \leftrightarrow j} y_{ij}$.
  - *end if*

Notice that this algorithm, like the diffusion algorithm, only requires communication for sending/receiving load to neighbouring processors, at Steps 2(c) or 2(i). There is no requirement for global communications, except when a convergence check is needed.

Table 3 illustrates the improved diffusion algorithm applied to the processor graph of Figure 2.

3.6 Convergence Analysis

The convergence rate of the improved diffusion algorithm can be easily analyzed using the definition of the polynomials p_k and the properties of the Chebyshev polynomials.

Using (23) and (29),

$$
\begin{aligned}
||e^{(k)}||^2 &\le \left(\max_{i=2}^{|V|} |p_k(\lambda_i)|^2\right) ||e^{(0)}||^2 \\
&\le \max_{\lambda_2 \le \lambda \le \lambda_{|V|}} |p_k(\lambda)|^2 \; ||e^{(0)}||^2 \\
&= \max_{\lambda_2 \le \lambda \le \lambda_{|V|}} \left|\frac{T_k(\xi(\lambda))}{T_k(\xi(0))}\right|^2 \; ||e^{(0)}||^2.
\end{aligned}
$$

Because $\xi(\lambda) \in [-1, 1]$ and $|\xi(0)| = |\lambda_2 + \lambda_{|V|}|/|\lambda_{|V|} - \lambda_2| > 1$, using the alternative expressions (27) and (28) for the Chebyshev polynomials,

$$
\begin{aligned}
||e^{(k)}|| &= \max_{\lambda_2 \le \lambda \le \lambda_{|V|}} \left|\frac{\cos(k \arccos(\xi(\lambda))}{\frac{1}{2}\left[\left(\xi(0) - \sqrt{\xi(0)^2 - 1}\right)^k + \left(\xi(0) + \sqrt{\xi(0)^2 - 1}\right)^k\right]}\right| ||e^{(0)}|| \\
&\le \left|\frac{2}{\left(\xi(0) - \sqrt{\xi(0)^2 - 1}\right)^k + \left(\xi(0) + \sqrt{\xi(0)^2 - 1}\right)^k}\right| ||e^{(0)}|| \\
&= \left|\frac{2\left(\xi(0) + \sqrt{\xi(0)^2 - 1}\right)^k}{1 + \left(\xi(0) + \sqrt{\xi(0)^2 - 1}\right)^{2k}}\right| ||e^{(0)}|| \\
&\le 2\left|\xi(0) + \sqrt{\xi(0)^2 - 1}\right|^k \; ||e^{(0)}||
\end{aligned}
$$

The error norm is reduced by approximately a factor of $\rho = |\xi(0) + \sqrt{\xi(0)^2 - 1}|$ per iteration, which is

$$
\begin{aligned}
\rho = \left|\xi(0) + \sqrt{\xi(0)^2 - 1}\right| &= \left|-\frac{\lambda_2 + \lambda_{|V|}}{\lambda_{|V|} - \lambda_2} + \sqrt{\left(\frac{\lambda_2 + \lambda_{|V|}}{\lambda_{|V|} - \lambda_2}\right)^2 - 1}\right| \\
&= \frac{\sqrt{\lambda_{|V|}} - \sqrt{\lambda_2}}{\sqrt{\lambda_{|V|}} + \sqrt{\lambda_2}} \\
&= \frac{\sqrt{cond} - 1}{\sqrt{cond} + 1},
\end{aligned}
$$

where $cond = \lambda_{|V|}/\lambda_2$ denotes the condition number of the weighted Laplacian matrix.

This average rate of convergence is better than that of the algorithm which uses the "best" polynomial (29) of degree one, that is,

$$l^{(k+1)} = \left(I - \frac{L}{\beta}\right) l^{(k)}. \tag{36}$$

Algorithm (36) has an average rate of convergence of

$$\rho_1 = \frac{\lambda_{|V|} - \lambda_2}{\lambda_{|V|} + \lambda_2} = \frac{cond - 1}{cond + 1} \geq \rho.$$

The diffusion algorithm (5) can be written as

$$l^{(k+1)} = (I - L)\, l^{(k)}.$$

It has therefore an average rate of convergence of

$$\rho_2 = \max\left\{|1 - \lambda_2|, |1 - \lambda_{|V|}|\right\}.$$

It can be proved that

$$\rho_2 \geq \rho_1 \geq \rho.$$

3.7 An Optimal Property

It was proved in Theorem 3 that the diffusion algorithm has a minimum property. Namely, the "flow" generated by the algorithm has the minimum weighted Euclidean norm. The same theorem is valid for the improved diffusion algorithm.

Theorem 4 *The "flow" generated by the improved diffusion algorithm (Algorithm CHEBY) minimizes $x^T W^{-1} x$, the scaled norm of the load migration, with the diagonal matrix W of size $|E| \times |E|$, consisting of the edge weights c_{ij} used in the algorithm.*

Proof The proof follows from that of Theorem 3, if one can prove that the amount of load transfer up to iteration k, $x^{(k)}$, converges to a vector x and furthermore, that there exists a vector d such that $x = WA^T d$ and $Ax = b$.

Consider the sequence of potentials $d^{(k)}$ of the algorithm. Denote $\bar{d}^{(k)} = d^{(k)} - (z^T d^{(k)})z/|V|$ the normalized potential vector (with a sum of zero), where z is the vector of all ones. From (34) one has

$$l^{(0)} - l^{(k)} = Ld^{(k)} = L\bar{d}^{(k)}. \tag{37}$$

Expand $\bar{d}^{(k)}$ using the normalized eigenvectors of L, because $\bar{d}^{(k)}$ is orthogonal to the eigenvector $u_1 = z$,

$$\bar{d}^{(k)} = \sum_{i=2}^{|V|} a_i^{(k)} u_i.$$

Substituting to (37) gives

$$l^{(0)} - l^{(k)} = \sum_{i=2}^{|V|} \lambda_i a_i^{(k)} u_i.$$

Thus because of (18),

$$||\bar{d}^{(k)}||^2 = \sum_{i=2}^{|V|} \left(a_i^{(k)}\right)^2 \leq ||l^{(0)} - l^{(k)}||^2/\lambda_2^2.$$

Since $l^{(k)}$ converged to the uniform load $\bar{l}z$, it follows that $||\bar{d}^{(k)}||$ is bounded. So there exists a subsequence $\{k_i\}$ such that

$$\bar{d}^{(k_i)} \to \bar{d}, \quad k_i \to \infty$$

Because of (37), $L\bar{d} = l^{(0)} - \bar{l}z$, so

$$L\left(\bar{d}^{(k)} - \bar{d}\right) = l^{(k)} - \bar{l}z \to 0.$$

It can now be proved that $\bar{d}^{(k)}$ converges to $\bar{d}$, as follows. The vector $\bar{d}^{(k)} - \bar{d}$ is orthogonal to the eigenvector $u_1 = z$. Expanding $\bar{d}^{(k)} - \bar{d}$ using the rest of the eigenvectors of L gives

$$\bar{d}^{(k)} - \bar{d} = \sum_{i=2}^{|V|} b_i^{(k)} u_i.$$

Multiplying both side with L, it follows that

$$\sum_{i=2}^{|V|} b_i^{(k)} \lambda_i u_i = L\left(\bar{d}^{(k)} - \bar{d}\right) \to 0.$$

Therefore

$$||\bar{d}^{(k)} - \bar{d}|| = \sum_{i=2}^{|V|} \left(b_i^{(k)}\right)^2 \le ||\sum_{i=2}^{|V|} b_i^{(k)} \lambda_i u_i||^2/\lambda_2^2 \to 0,$$

or $\bar{d}^{(k)} \to \bar{d}$

Having proved that $\bar{d}^{(k)}$ converges, it follows from (33) that $x^{(k)}$, the amount of transfer up to iteration k, also converges

$$x^{(k)} = WA^T d^{(k)} = WA^T \bar{d}^{(k)} \to WA^T \bar{d} = x$$

Furthermore,

$$Ax = AWA^T \bar{d} = L\bar{d} = l^{(0)} - \bar{l}z = b.$$

□

4 Numerical Experiments

In this section the improved diffusion algorithm is compared with the diffusion algorithm and the fast global algorithm.

It was found in [9] that the global algorithm is in general faster than the diffusion algorithm on Cray T3D, using up to 256 processors. One is interested in knowing how the algorithms scale beyond hundred of processors. For this reason the algorithms were implemented on a serial computer to simulate the result of these algorithms on up to a few thousand processors. Algorithms were compared by the number of iterations they took to converge to the same criterion.

The convergence criterion for all the algorithms was

$$\text{load imbalance} \;=\; \max_{i\in V} \left\{ \frac{l_i - \bar{l}}{\bar{l}} \right\} \;<\; \epsilon,$$

where ϵ is the maximum load imbalance tolerable overall all vertices. Two tolerances were used, namely $\epsilon = 0.1$ and $\epsilon = 0.01$, representing a loose and a tight tolerance to the load imbalance.

Test results for algorithms on random graphs will be given. The processor graphs resulted from unstructured mesh applications usually do not have a uniform structure, and can be represented reasonably well by random graphs. However the algorithms were also tested on uniform graphs such as meshes, and the results were consistent with those on random graphs and are thus not presented.

A random graph generator was written. Given $|V|$ vertices, the generator randomly linked vertices until the average degree of the graph reached the preset value. The graph was then checked for its connectivity, and extra edges were added if the graph was found to be disconnected, in which case the final degree of the graph could be slightly larger than the preset value due to the extra edges. The resulting graphs were used to test the algorithms. The preset values for the average degree were 1, 3, 5, 7 and 9. For each value, random graphs of size 500, 1000 and 2000 were generated.

The initial load on each vertex was set according to two schemes.

- *Random initial load*: the load on each vertex was set to be $200 * ran$, with ran a random number. Thus the average load was about 100 and initial load imbalance was around $(200 - 100)/100 = 1$.

- *Step-Function initial scheme*: the load on one vertex was set to $100|V|$, and the load on all other vertices was set to 0. Thus the average load was 100 and initial load imbalance was $(100|V| - 100)/|V| = |V| - 1$.

These initial load schemes are meant to test the ability of the algorithms on two typical kinds of load imbalance: those caused by the fluctuation of load across the processors, and those caused by a sudden change of load on one processor.

Notice that all of the algorithms are invariant to a uniform addition or subtraction of a constant to the initial load, it is therefore not necessary to test other related initial load schemes, such as $l_i = 100 + 200 * ran$

Five algorithms were tested, and the results are listed in Tables 4-7. The global algorithm is denoted as CG, the diffusion algorithm is denoted as DIFF and the improved diffusion algorithm is denoted as CHEBY. Two restarted versions of CHEBY were also tested, they are denoted as CHEBY1 and CHEBY10, which are CHEBY but restarted every 1 and 10 iterations respectively. The reason for testing the restarted algorithms is as follows. The diffusion algorithm has the advantage that it is able to handle load that changes while the diffusion algorithm is being applied. The improved diffusion algorithm CHEBY is not designed to cope with such a situation. However it is possible to use a restarted version of the algorithm to take into account the load changes. For this reason one is interested in knowing how such restarted algorithms compare with the diffusion algorithm.

The coefficient c_{ij} for the weighted Laplacian was chosen as in equation (7). All algorithms were run to convergence, the accumulated amount of load transfer was then use to migrate loads to neighbouring processors.

In unstructured mesh based applications, it is important to decide not only the amount of load to be transferred, but also which mesh nodes of the subdomains are to be migrated. This is beyond the scope of this paper and we refer to [12] for further details.

If a processor needed to send out a total of OUT units of load, and this was greater than the amount of load it currently had (denote this amount as ON), then the actual amount sent out was scaled by a factor of ON/OUT, rounding to an integer. The number of iterations it took for this process to converge is given in the last column of the tables, under the heading MIGRATE. Because of the minimization property of CG, DIFF and CHEBY, the three algorithms produce exactly the same amount of accumulated load transfer (subject to numerical accuracy). So the number of iterations for the MIGRATE process is the same for all three algorithms. It is also easy to prove, following the same line as the proof of Theorem 4, that the restarted CHEBY algorithms have the minimization property as well, and thus the same number of iterations for the MIGRATE process.

A disadvantage of improved diffusion algorithms is that it is necessary to know the smallest and largest positive eigenvalues of the weighted Laplacian matrix. It is therefore only suitable if the processor graph is known in advance and does not change, and many dynamic load balancing steps are to be carried out, thus off-setting the one-off cost of computing the eigenvalues. The two extreme eigenvalues were calculated using LAPACK for $|V| \leq 1000$. For $|V| > 1000$, the memory requirement of storing the whole matrix became excessive. So a Lanczos algorithm was used for calculating the smallest positive eigenvalue and a Power algorithm was used for the largest eigenvalue. Both algorithms were iterative and required the storage of only a few vectors. Both algorithms were terminated at a tolerance of 10^{-5}. It was found however that the Chebyshev iterative process was sensitive to the accuracy of the eigenvalues. In particular, slight over prediction of λ_2 or under prediction of $\lambda_{|V|}$ could cause the process to diverge. For this reason when the iterative algorithms were used to calculate the eigenvalues, the calculated eigenvalues, λ_2 and $\lambda_{|V|}$, were modified slightly to $0.95\lambda_2$ and $1.05\lambda_{|V|}$ respectively.

It is seen from the tables that in terms of the number of iterations, all algorithms performed better as the connectivity of the graph increased. When the average degree was higher than 5, all algorithms converged in less than 280 iterations. Overall, algorithm CG was the best for all the graphs. The improved diffusion algorithm CHEBY out-performed the diffusion algorithm DIFF.

From Table 4, when applied to a random initial load, with a loose tolerance of $\epsilon = 0.1$, the diffusion algorithm DIFF did not perform too badly compared with CG and CHEBY. Reading this table alone might suggest that the more sophisticated algorithms may not be necessary. The reality is that DIFF is a stationary iterative algorithm, and like other stationary type iterative algorithms (e.g., Jacobi, Gauss-Seidel methods for linear systems), it is good at reducing the high frequency errors (see [2] for definition), but very slow in reducing the low frequency errors. When the initial load is random, neighbouring vertices tend to have quite different loads, but the average load over a neighbourhood is likely to be close to the average over the whole graph. Thus there is a large component of high frequency error, and as a result the diffusion algorithm can quickly damp out this error. If, however, the initial error does not oscillate as much as in the case of random initial load, the diffusion algorithm should take a long time to achieve even a loose load balance. This is pseen in the case of a step-function initial load, in Table 6. Likewise, when the convergence criterion is tighter, DIFF also suffers, as seen from Table 5 and Table 7.

The restarted algorithms CHEBY10 and CHEBY1 were in general better than DIFF, although the two restarted algorithms behaved erratically on three occasions when applied to problems with random initial load. For example, CHEBY10 took 61023 iterations on the graph of 2000 vertices with an average degree of 2. The reason is not clear.

Comparing CHEBY with CG, it is seen that the former took around twice the number of iterations on problems with random initial loads, and around 3 times as long on problems with step-function initial loads. It has the advantage that global communication is not needed. CHEBY is thus a viable alternative to CG when the cost of eigenvalue calculations can be off-set.

5 Conclusions

In this paper an improved diffusion algorithm is derived based on Chebyshev polynomials. An iterative procedure to calculate the potential is also given.

Convergence analysis and numerical tests show that the improved diffusion algorithm is significantly faster than the diffusion algorithm, at the extra cost of the calculation of two extreme eigenvalues. It is also shown that the both the diffusion algorithm and the improved diffusion algorithm have the minimum "flow" property. This means that when using the same edge weights, both algorithms as well as the global algorithm generate the same minimum "flow" (subject to numerical error).

The improved diffusion algorithm is a viable alternative to the global algorithm (CG) when global communication is not desirable, and when the cost of eigenvalue calculations is not significant, e.g., when the processor graph is known in advance and does not change, and many dynamic load balancing steps are to be carried out, thus off-setting the one-off cost of computing the eigenvalues. The restarted version of the algorithm may also be used for applications where the load changes rapidly.

Acknowledgments The authors would like to thank the referees for very constructive and detailed review of the paper.

References

[1] J. E. Boillat, Load balancing and Poisson equation in a graph. *Concurrency: Practice and Experience* **2** (1990) 289-313.

[2] W. L. Briggs, *A Multigrid Tutorial* (SIAM, Philadelphia, 1987).

[3] G. Cybenko, Dynamic load balancing for distributed memory multiprocessors. *J. Parallel Distrib. Comput.* **7** (1989) 279-301.

[4] R. Fletcher, *Practical Methods of Optimization* (John Wiley and Sons, Chichester, 1987).

[5] G. H. Golub and C. F. Van Loan, *Matrix Computations* (Johns Hopkins University Press, Baltimore, 1981).

[6] A. Heirich and S. Tayler, A parabolic load balancing method, Technical Report, Caltech Computer Science Department, Caltech-CS-TR-94-13, 1994.

[7] K. A. Hoffmann, *Computational Fluid Dynamics* (Engineering Education System, Taxas, 1989).

[8] G. Horton, A multi-level diffusion method for dynamic load balancing, *Parallel Computing* **9** (1993) 209-218.

[9] Y. F. Hu, R. J. Blake and D. R. Emerson, An optimal migration algorithm for dynamic load balancing, *Concurrency: Practice and Experience* **10** (1998) 467-483.

[10] A. Pothen, D. H. Simon and K. P. Liou, Partitioning sparse matrices with eigenvectors of graphs, *SIAM J. Matrix Anal. Appl.* **11** (1990) 430-452.

[11] J. Song, A partially asynchronous and iterative algorithm for distributed load balancing, *Parallel Computing* **20** (1994) 853-868.

[12] C. Walshaw, M. Cross and M. G. Everett, Parallel dynamic graph partitioning for adaptive unstructured meshes, *J. Parallel Distrib. Comput.* **47** (1997) 102-108.

[13] C. Z. Xu, and F. C. M. Lau, Analysis of the generalized dimension exchange method for dynamic load balancing, *J. Parallel Distrib. Comput.* **16** (1992) 385-393.

[14] C. Z. Xu, F. C. M. Lau, B. Monien and R. Lüling, Nearest-neighbor algorithms for load balancing in parallel computers, *Concurrency: Practice and Experience* **7** (1995) 707-736.

[15] D. M. Young, *Iterative solution of large linear systems* (Academic Press, NewYork, 1971).

Table 1: Applying the diffusion algorithm on the graph of Figure 2: the load on each processor at each iteration

Iteration	1	2	3	4	5	6	7	8
0	25.00	15.00	15.00	15.00	15.00	15.00	15.00	15.00
1	22.50	17.50	15.00	15.00	15.00	15.00	15.00	15.00
2	21.25	17.63	15.00	15.63	15.00	15.50	15.00	15.00
3	20.34	17.61	15.21	15.92	15.10	15.63	15.10	15.10
4	19.66	17.47	15.41	16.10	15.24	15.71	15.21	15.21
5	19.11	17.32	15.58	16.21	15.39	15.77	15.31	15.31
6	18.67	17.18	15.73	16.28	15.53	15.82	15.40	15.40
7	18.29	17.05	15.85	16.32	15.65	15.86	15.48	15.48
8	17.98	16.94	15.94	16.35	15.76	15.91	15.56	15.56
9	17.72	16.85	16.02	16.36	15.85	15.95	15.63	15.63
10	17.51	16.76	16.08	16.37	15.92	15.98	15.69	15.69
11	17.32	16.69	16.12	16.37	15.99	16.01	15.75	15.75
12	17.16	16.63	16.16	16.37	16.04	16.04	15.80	15.80
13	17.03	16.58	16.19	16.36	16.08	16.06	15.85	15.85
14	16.92	16.54	16.21	16.36	16.11	16.08	15.89	15.89
15	16.82	16.50	16.23	16.35	16.14	16.10	15.93	15.93
16	16.74	16.46	16.24	16.35	16.16	16.12	15.96	15.96
17	16.67	16.43	16.25	16.34	16.18	16.13	16.00	16.00
18	16.61	16.41	16.26	16.33	16.19	16.15	16.02	16.02
19	16.56	16.39	16.26	16.33	16.21	16.16	16.05	16.05
20	16.52	16.37	16.26	16.32	16.21	16.17	16.07	16.07
21	16.48	16.36	16.27	16.31	16.22	16.18	16.09	16.09
22	16.45	16.34	16.27	16.31	16.23	16.19	16.11	16.11
23	16.42	16.33	16.27	16.30	16.23	16.19	16.12	16.12

Table 2: Applying the global algorithm on the graph of Figure 2: the load on each processor at each iteration

Iteration	1	2	3	4	5	6	7	8
0	25.00	15.00	15.00	15.00	15.00	15.00	15.00	15.00
1	16.20	23.76	14.77	15.17	15.30	14.51	15.15	15.15
2	16.22	16.23	15.03	19.19	15.12	17.85	15.18	15.18
3	16.25	16.26	17.27	16.51	16.00	15.75	15.99	15.99
4	16.26	16.22	16.08	16.54	17.02	15.78	16.05	16.05
5	16.24	16.26	16.37	16.03	16.39	16.63	16.04	16.04
6	16.25	16.25	16.25	16.25	16.25	16.25	16.25	16.25

Table 3: Applying the improved diffusion algorithm on the graph of Figure 2: the load on each processor at each iteration

Iteration	1	2	3	4	5	6	7	8
0	25.00	15.00	15.00	15.00	15.00	15.00	15.00	15.00
1	21.02	18.98	15.00	15.00	15.00	15.00	15.00	15.00
2	17.84	17.90	15.00	17.37	15.00	16.89	15.00	15.00
3	16.83	16.81	16.67	16.76	15.80	15.54	15.80	15.80
4	16.53	15.97	16.61	16.55	16.51	15.99	15.92	15.92
5	16.17	16.33	16.49	16.24	16.56	16.25	15.98	15.98
6	16.15	16.27	16.34	16.37	16.40	16.26	16.11	16.11
7	16.21	16.24	16.36	16.33	16.26	16.20	16.20	16.20

Table 4: Number of iterations for the 5 algorithms, with random initial load, $\epsilon = 0.1$

$\|V\|$	degree	diameter	CG	CHEBY	CHEBY10	CHEBY1	DIFF	MIGRATE
500	2	305	273	448	2927	885	1329	8
500	3	16	17	23	29	54	80	2
500	5	9	7	10	10	30	42	2
500	7	6	4	8	8	11	13	2
500	9	4	4	6	6	11	14	2
1000	2	612	585	871	13420	108523	1773	12
1000	3	29	24	43	65	55	80	3
1000	5	9	8	10	10	27	38	2
1000	7	6	6	8	8	17	23	2
1000	9	5	4	8	8	19	26	2
2000	2	1138	1221	1809	61023	6950	9955	43
2000	3	35	32	57	102	250	353	4
2000	5	14	13	18	20	83	116	2
2000	7	7	5	9	9	20	26	2
2000	9	7	5	9	9	16	22	2

Table 5: Number of iterations for the 5 algorithms, with random initial load, $\epsilon = 0.01$

$\|V\|$	degree	diameter	CG	CHEBY	CHEBY10	CHEBY1	DIFF	MIGRATE
500	2	305	361	774	11327	50721	28299	8
500	3	16	26	39	52	266	396	2
500	5	9	11	16	17	59	84	2
500	7	6	7	13	12	30	41	2
500	9	4	6	9	9	20	27	2
1000	2	612	733	1505	47030	-	134920	12
1000	3	29	34	71	139	758	1118	3
1000	5	9	13	16	17	59	85	2
1000	7	6	11	13	13	37	52	2
1000	9	5	7	13	13	38	54	2
2000	2	1138	1392	3124	-	-	-	43
2000	3	35	40	94	232	1415	2006	4
2000	5	14	18	30	37	199	280	2
2000	7	7	9	14	15	42	57	2
2000	9	7	8	15	16	36	49	2

Table 6: Number of iterations for the 5 algorithms, with step-function initial load, $\epsilon = 0.1$

$\lvert V \rvert$	degree	diameter	CG	CHEBY	CHEBY10	CHEBY1	DIFF	MIGRATE
500	2	305	370	1064	18610	117823	177123	234
500	3	16	28	52	84	259	385	15
500	5	9	12	17	20	43	47	7
500	7	6	8	17	18	50	36	6
500	9	4	8	13	12	22	23	5
1000	2	612	627	2200	40073	115278	173928	377
1000	3	29	40	105	250	1607	1525	24
1000	5	9	15	22	23	47	67	8
1000	7	6	10	17	20	31	39	7
1000	9	5	9	13	14	29	31	6
2000	2	1138	1426	4914	199966	-	-	1021
2000	3	35	52	154	458	1538	2181	34
2000	5	14	15	27	33	59	82	8
2000	7	7	14	24	28	46	63	7
2000	9	7	9	19	22	30	40	6

Table 7: Number of iterations for the 5 algorithms, with step-function initial load, $\epsilon = 0.01$

$\|V\|$	degree	diameter	CG	CHEBY	CHEBY10	CHEBY1	DIFF	MIGRATE
500	2	305	379	1384	26919	-	-	234
500	3	16	36	62	108	464	691	15
500	5	9	15	22	26	66	82	7
500	7	6	12	21	24	71	67	6
500	9	4	10	16	17	33	41	5
1000	2	612	679	2767	72349	-	-	377
1000	3	29	47	138	347	2515	2879	24
1000	5	9	18	28	30	77	111	8
1000	7	6	13	21	26	50	63	7
1000	9	5	10	17	20	40	43	6
2000	2	1138	1463	6104	-	-	-	1021
2000	3	35	61	188	598	2887	4094	34
2000	5	14	19	34	42	97	134	8
2000	7	7	16	30	36	83	115	7
2000	9	7	12	24	27	53	72	6