Chapter 4

Load Balancing for Variable Conductance Diffusion

Don't go into the Niagara River, and if you do, don't take a right.
— Capt. Peter Wiles, Sr., "Erie Canal Guide"

4.1 Variable Conductance Diffusion

4.1.1 Background

The field of image processing covers algorithms and computations that process digital representations of images. Often these images are produced by video cameras or similar devices, but images can also be produced by sonar equipment, or by medical imaging devices such as Computerized Tomography scanners or Magnetic Resonance Imaging equipment, which produce images of the real world that could never be seen by the unaided human eye.

The purpose of image processing is to improve the intelligibility of an image, its aesthetic appearance, or both. Images of the real world often contain noise due to the imperfection of the recording apparatus; Variable Conductance Diffusion is designed to eliminate some forms of noise, improving both the intelligibility and the aesthetic appearance of an image.

An image is a two dimensional grid of pixel values, with each pixel value uniquely specifying a color or shade of grey for a point in the image. For

computing, color images can be treated as three grey images rather than one composite image.

Some types of noise affect individual pixels, so a single dot in the image will have a value that is completely different from those surrounding it, even though it belongs to the same part of the image. Such *speckle* noise can be removed by diffusion algorithms such as Variable Conductance Diffusion. In diffusion, the value of each pixel is repeatedly modified in the direction of the average value of its neighbors. An image that has been diffused is blurred with respect to the original image, because every feature and edge has been "washed out" over several pixels. This blurring also diffuses any pixel noise, since the total energy of the speckle noise is no longer concentrated on a single pixel but distributed over several pixels, where it is no longer as visible.

4.1.2 Algorithm

Variable Conductance Diffusion [72] removes speckle noise from the image, but is unlike plain diffusion in that the noise is removed without otherwise blurring the image. In particular, the magnitude by which each pixel is modified in the direction of the average value of its neighbors is inversely proportional (by some function *phi*) to the gradient of the image in the neighborhood of the pixel. One proportionality function *phi* that has been used is the scaled negative exponential of the square of the difference in pixel values, $ve^{-v^2/2}$, with v the difference in value between each pair of neighboring pixels. The result of Variable Conductance Diffusion is an image that has been smoothed everywhere except near sharp edges. Figure 4.1 gives the algorithm for Variable Conductance Diffusion.

The actual implementation of Variable Conductance Diffusion that this work is based on was written in Modula-2 by Richard de Moliner [27] at ETH Zurich, in Switzerland, where it was used for edge detection in images. This program was converted to C by Robert Cromartie of UNC. The program includes an optimization that saves partial results in the computation of the weighted gradient so that the function *phi* is only invoked four times for each pixel, instead of the eight that would be required otherwise.

Variable Conductance Diffusion algorithm for image I of size $n \times m$
The two parameters of the computation are:
k the coefficient by which the gradient is scaled
dt the speed of convergence

```
foreach i in 1...n
  foreach j in 1...m
    new I[i,j] ← I[i,j] + k × dt × δ
where
  δ ← δ0 + δ1 + δ2 + δ3
  δ0 ← phi(I[i+1,j] − I[i,j]) − phi(I[i,j] − I[i−1,j])
  δ1 ← phirt(I[i+1,j+1] − I[i,j]) − phirt(I[i,j] − I[i−1,j−1])
  δ2 ← phi(I[i,j+1] − I[i,j]) − phi(I[i,j] − I[i,j−1])
  δ3 ← phirt(I[i−1,j+1] − I[i,j]) − phirt(I[i,j] − I[i+1,j−1])
  phi(v) ← x e^−(x^2/2) where x ← v/k
  phirt(v) ← (x e^−(x^2/2))/√2 where x ← v/(k√2)
```

Figure 4.1: Algorithm for Variable Conductance Diffusion

4.2 Parallel Implementation

4.2.1 Decomposition and Communication

The author's parallel implementation of Variable Conductance Diffusion is based on the above-mentioned C version by Robert Cromartie. The implementation is for the MasPar MP-1, a mesh connected massively parallel[1] SIMD[2] computer which can be programmed in MPL, a data parallel variant of the C language. For communication, the MP-1 provides both a general purpose router and a two dimensional mesh, and MPL provides efficient access to both.

The router is faster for sending small amounts of data long distances, but the mesh is faster for sending large amounts of data short distances; the diffusion program only requires communication among neighboring pixels, and

[1] *Massively parallel* is often used to refer to computers with 1000 or more processors; the model currently installed at UNC has 4096 processors.

[2] *Single Instruction stream, Multiple Data stream* describes a parallel computer with centralized control of distributed parallel processors.

for efficiency only uses the mesh. The image is subdivided hierarchically into tiles and the program performs the diffusion independently on each tile, after copying the pixels from the edge of each of the four neighboring processors and copying the corner pixels from the four corner processors. In a true mesh, the corner copying operation would require communication over a distance of two links, but the MP-1 provides support for direct communication along diagonals, so this is a local operation; if the diagonal connections were missing, this would still be a local operation, though the neighborhood over which communication takes place would become a neighborhood of size 2 instead of size 1. OMS decompositions preserve such neighborhoods, so even in the absence of diagonal connections, no communication over distance greater than two is required. Variable Conductance Diffusion can therefore be thought of as an algorithm requiring only local communication.

4.2.2 Data

For typical image sizes of 256×256 or 512×512 pixels, each processor on a 64×64 processor MP-1 such as the one at the University of North Carolina does the diffusion for a 4×4 or an 8×8 tile of the image. The computation uses 6 floating point values for each pixel: one for the source image, one for the destination image, and four for the intermediate results that optimize the gradient computation. Since each pixel therefore takes up 24 bytes and the configuration of the MP-1 at the University of North Carolina provides 16K bytes of memory for each processor, the maximum size square tile that is a power of two that can be assigned to each processor is 16×16, corresponding to a 1024×1024-pixel image and requiring 6144 bytes in each processor for storing the image. A tile that were twice as large in each dimension would require four times as much storage, and that is more than can be stored in one processor. The algorithm for Variable Conductance Diffusion could be changed to not use such intermediate storage, but it seemed more useful to gather data about the performance of Scan Directed Load Balancing on an example of a computation that is actually used in practice.

The hierarchical subdivision of the image into contiguous rectangular tiles minimizes the time required for communication; in this implementation, significantly more time is spent computing the diffusion than communicating pixel values.

4.2.3 Input Data

The diffusion program has been run on 33 medical and test image files of sizes 512×512 pixels (large), 400×448 pixels (medium), and 256×256 pixels (small). For these images each processor can store tiles that are two to four times larger, in each dimension, than a tile from a uniformly decomposed image; this limited buffering flexibility both provides some space for redistributing load to processors, and raises the possibility of buffer overflows. A typical image is shown in Figure 4.2; the result of performing Variable Conductance Diffusion on it with a particular choice of parameters is shown in Figure 4.3.

The contours in the original image have disappeared and the image is now more realistic, so it is easier to recognize features and distinguish detail, even though strictly speaking some information has been lost. The contours are not speckle noise and could have been removed by other algorithms, but this example shows that Variable Conductance Diffusion can achieve good results without any prior knowledge about the image being processed. Notice how the edges of the orbits and of all the other bones remain sharply defined, whereas the unwanted fine detail has been diffused (blurred) to the point of being unrecognizable.

4.3 Load Balancing for Diffusion

4.3.1 Applicability for Load Balancing

The parallel Variable Conductance Diffusion program is a good test bed for Scan Directed Load Balancing. Diffusion requires nothing but local communication on the MP-1 (communication to distance no greater than two for purely rectangular mesh connected systems), so communication is relatively fast and any speedup in computing time generally results in an overall speedup for the entire program. Variable Conductance Diffusion is a good candidate for load balancing, as are many other diffusion algorithms, because different pixels require different numbers of averaging iterations. Diffusion, for instance, does not substantially the values of pixels that are already close to the average of their neighbors; an optimized version of the diffusion algorithm in Figure 4.1 that takes advantage of this to avoid some work is shown in Figure 4.4. If a pixel is within a threshold ϵ of the weighted average of its

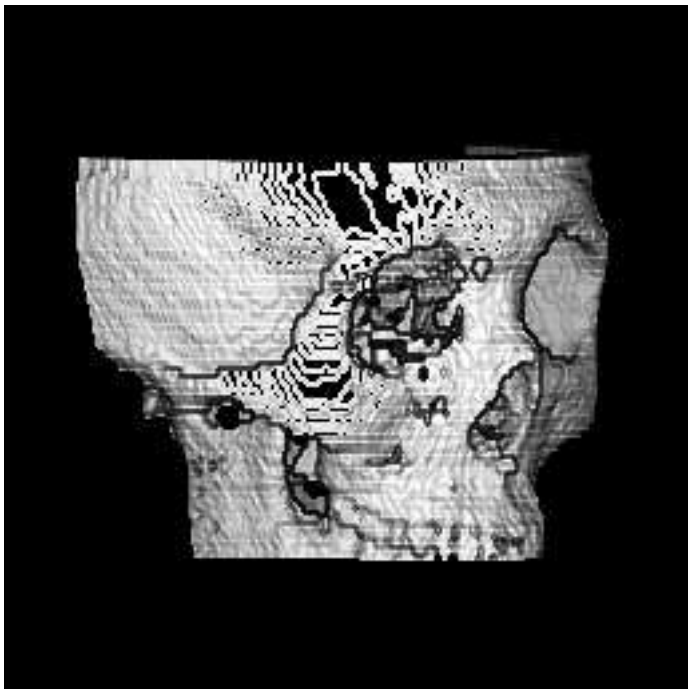

Figure 4.2: The Original Image

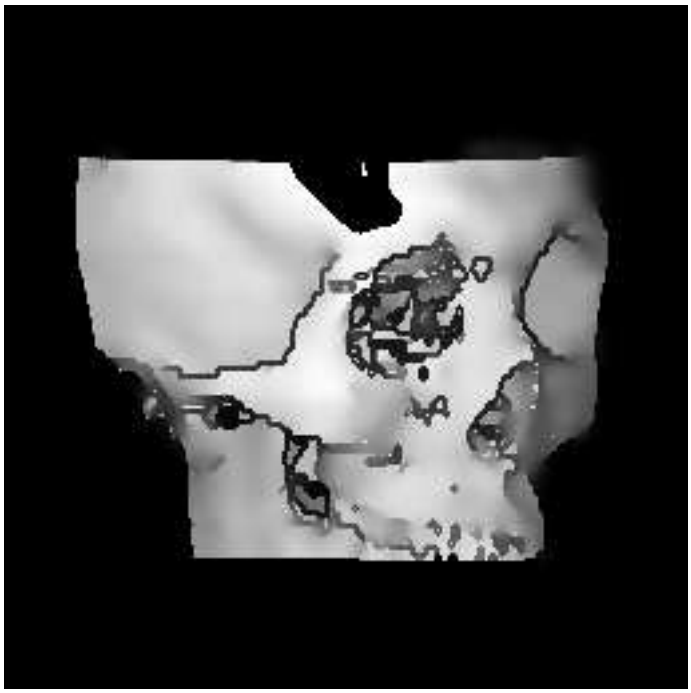

Figure 4.3: The Image Produced by Variable Conductance Diffusion

An Efficient algorithm for Variable Conductance Diffusion on image I of size $n \times m$
The boolean array A is *true* for all those elements of the image which are still diffusing and for their neighbors.

```
foreach i in 1...n
  foreach j in 1...m
    if (A[i,j]) then
      newI[i,j] ← I[i,j] + k × dt × δ
      A'[i,j] ← δ > ε
    else
      A'[i,j] ← false
foreach i in 1...n
  foreach j in 1...m
    newA[i,j] ← A'[i,j] ∨ new(i,j)
where
  new(i,j) ← newup(i,j) ∨ newdn(i,j) ∨ newside(i,j)
  newup(i,j) ← A'[i-1,j+1] ∨ A'[i,j+1] ∨ A'[i+1,j+1]
  newdn(i,j) ← A'[i-1,j-1] ∨ A'[i,j-1] ∨ A'[i+1,j-1]
  newside(i,j) ← A'[i-1,j] ∨ A'[i+1,j]
  δ ← ...
```

Figure 4.4: Optimized Algorithm for Variable Conductance Diffusion

neighbors, it becomes inactive and requires no further computation, unless and until one of its neighbors changes its value. A pixel that is in a noisy area may need many iterations to converge to a steady-state value. Pixels at the edge of a noisy area may converge quickly, but then need further adjustments as their neighbors' values keep changing. Because these different cases cannot be distinguished until run time, when the image is available, and because the pixel may reach a steady-state value then need adjustments several times during a diffusion, a dynamic load balancing algorithm is likely to give better performance than a static load balancing algorithm.

Each image is a two dimensional array of pixels, so it makes sense to talk about rows and columns of pixels. Since the processors on the MP-1 are arranged in a two dimensional mesh, it is also useful to talk about rows and columns of processors. The load is balanced without violating the OMS (Orthogonality, Monotonicity, and Surjectivity) constraint, so every processor in a row has a tile of the same height as every other processor in the same row, and every processor in a column has a tile with the same width as every other processor in the same column. However, the height need not be the same for different rows, and the width need not be the same for different columns.

4.3.2 Data Structures

Load balancing a purely SIMD computer, in which each processor has to access the same memory location as every other processor, would require that all the points to be processed, the *active* points, be stored at the same location in the memory of every processor. Fortunately the MP-1 supports *local indirect addressing*, which allows each processor to follow a local pointer into its own memory. This flexibility means not only that the algorithms of Section 3.4.4 can be used, but that simpler methods are also applicable. The Variable Conductance Diffusion program that does load balancing, `bedd`, stores all points for each processor in row major order in a linear array of memory. The stride with which rows are stored in memory is the same for all processors, so all processors that have a data element D_G with local index $G = [r, c]$ store that element at the local address $r \times stride + c$. This uniform addressing scheme makes it easier for one processor to access the data of another, which is useful both for the transfer of edge pixels in Variable Conductance Diffusion and in the implementation of `put()` and `get()`.

An array of booleans (the *activity array*) on each processor is used to keep

track of active pixels on each iteration of the main loop. Pixels are active if and only if in the last cycle they or their neighbors changed their value by more than a given threshold. Only pixels which are active recompute their value. All processors compute the change in value for one pixel in purely SIMD fashion, then each searches through its activity array of booleans until it finds its next active pixel. Keeping a list of active pixels might be more efficient, but the small size of the tiles in each processor, and the coding and execution overhead required to maintain lists of active data points, makes the activity array strategy more desirable for this application.

4.3.3 Iteration Step

Approximately every five diffusion cycles, the main loop calls the load balancing routine `bal`. The number of diffusion cycles between calls to `bal` is parametrized, but for most images an interval of five to ten diffusion cycles between load balancing cycles seems to give good results. This is discussed further in Section 4.5.

The load balancing routine uses the processor load values as input to the algorithm of Figure 3.13. For experimental purposes, the load balancing algorithm can balance in either dimension, in both dimensions, or can alternate the dimension in which it balances every time `bal` is called. Following the algorithm of Figure 3.13, `bal` first computes the activity flows F_i, then performs the shifts.

4.4 Measuring Performance

For each image, the diffusion algorithm running without load balancing was compared to the diffusion algorithm with load balancing, using two distinct measures of performance. As used here, performance is inversely related to cost, which is related to the time needed to complete the computation.

One measure of cost averages, over all the diffusion cycles, the maximum load on any given processor. This *load* measure is an indication of the algorithm's effectiveness in reducing the load from the computation, and would correspond to the speedup if the load balancing could be done at no cost. The second measure of cost is the time required to complete the diffusion. This *time* measure gives an indication of the overhead required for the algorithm

and is therefore useful in judging the overall effectiveness of this implementation of Scan Directed Load Balancing. The time measure is affected by details of the implementation, and is not as useful as the load measure in judging the potential of the algorithm as a whole. The two measures are presented more formally in the following definitions.

Definition 22 *A* step *in the computation is a repeated portion of the computation during which processors need not be synchronized. All processors begin each step synchronously, and synchronize again at the end of the step.*

Load Balancing works by predicting the load for the next step from the activity of data points during the current step. In what follows, it is assumed that the computation has T steps, and that the majority of time required for the computation is spent executing the T steps.

Definition 23 *The* load cost *is the average over the time steps of the maximum load on any processor:*

$$Load = \frac{1}{T} \sum_{t<T} max_I(A_I^t)$$

Here the load A_I^t is the load at step t for processor P_I, and is proportional to the time required by P_I to complete the step.

The load cost gives a good indication of the potential speedup of Scan Directed Load Balancing for a given computation; different implementations of Scan Directed Load Balancing may be more or less effective at translating the improvement in load into savings in time.

Definition 24 *The* time cost *is the total time required for all the steps of a computation:*

$$Time = \sum_{t<T} time(t)$$

The time cost gives a good indication of the effectiveness of a particular implementation of load balancing in speeding up a given problem. The advantage of the time cost over the load cost is that the overhead of load balancing is taken into account in computing the total speedup due to Scan Directed Load Balancing, so the time cost is more indicative of the speedups

that can actually be achieved. The advantage of the load cost over the time cost is that the load cost only measures the improvement in load balance and is independent of implementation details. With either cost measure, the speedup is defined as one less than the ratio of the cost without load balancing to the cost when Scan Directed Load Balancing is used. Note that a negative speedup is a slowdown.

Definition 25 *A* cost function $c(n)$ *is a function int* $\rightarrow$ *real which gives a cost of executing a computation with* T *steps and balancing the load* n *times. Normally,* $n \leq T$.

A computation that runs without load balancing has a cost $c(0)$, one that load balances 5 times has a cost $c(5)$. The Variable Conductance Diffusion computation described here was run each time for $T = 50$ steps, with n varied from $n = 0$ to $n = 50$. Note that the cost function depends of the initial distribution μ_0.

Both the load cost and the time cost are cost functions, though the argument n of the cost function is not shown explicitly in the definition.

Definition 26 *The* speedup $Sp(n)$ *given by load balancing* n *times is one less than the ratio of the cost of not load balancing the computation versus the cost of load balancing.*

$$Sp(n) = \frac{c(0)}{c(n)} - 1$$

This definition of speedup is analogous to the one on page 6 of Hennessy and Patterson's text on Computer Architecture [39].

The next section presents the speedup obtained by Scan Directed Load Balancing on Variable Conductance Diffusion run on the medical images described above.

4.5 Results

4.5.1 Test Procedure

To test the performance of Scan Directed Load Balancing, Variable Conductance Diffusion was run both with and without load balancing on each

image. Twenty runs were done on each image using load balancing, to test four different options of dimension selection each with five different frequencies of load balancing. Both rows and columns were balanced, or only rows, or only columns, or both rows and columns but on alternate calls to `bal`. Load balancing was done once every diffusion step, every other step, every 5th, 10th, or 20th diffusion step. The parameters for Variable Conductance Diffusion, including the cutoff threshold for edge detection, were unchanged over all the runs, and in fact the load-balanced results are identical to those produced without load balancing.

4.5.2 Average Performance

The speedup given by load balancing for the 33 test images is shown in Figure 4.5. The graphs plot decreasing frequency of load balancing on the horizontal axis and increasing speedup on the vertical axis. The top graph plots the speedup in time given by balancing the load every x iterations of Variable Conductance Diffusion, where x is plotted along the horizontal axis. In this figure, the *average* speedup, marked by the numeric values, is uniformly negative (a slowdown), which means the load balancing is not, on average, being effective at speeding up Variable Conductance Diffusion. However, some of the points, representing the speedup obtained on individual images, correspond to a positive speedup; for these images, Scan Directed Load Balancing is effective at decreasing execution time.

The lower graph of Figure 4.5 plots load improvement against decreasing frequency of load balancing. All of the averages and most of the individual images have a positive load improvement, which means that load balancing decreased the load of most heavily loaded processor for most images. This is expected, but contrasts with the effective slowdown in time caused by load balancing, which is undesirable.

The average, over all images, of the speedup is printed in approximately the position corresponding to the average speedup. As can be seen, there is considerable variation in the performance on Scan Directed Load Balancing for different images, especially in the time cost. The speedup is better for the load cost, since this measure ignores the overhead of load balancing. In practice, the time speedup is negligible and many images actually take longer to diffuse when using load balancing. On the other hand, the load improvement is considerable, which indicates that a faster implementation

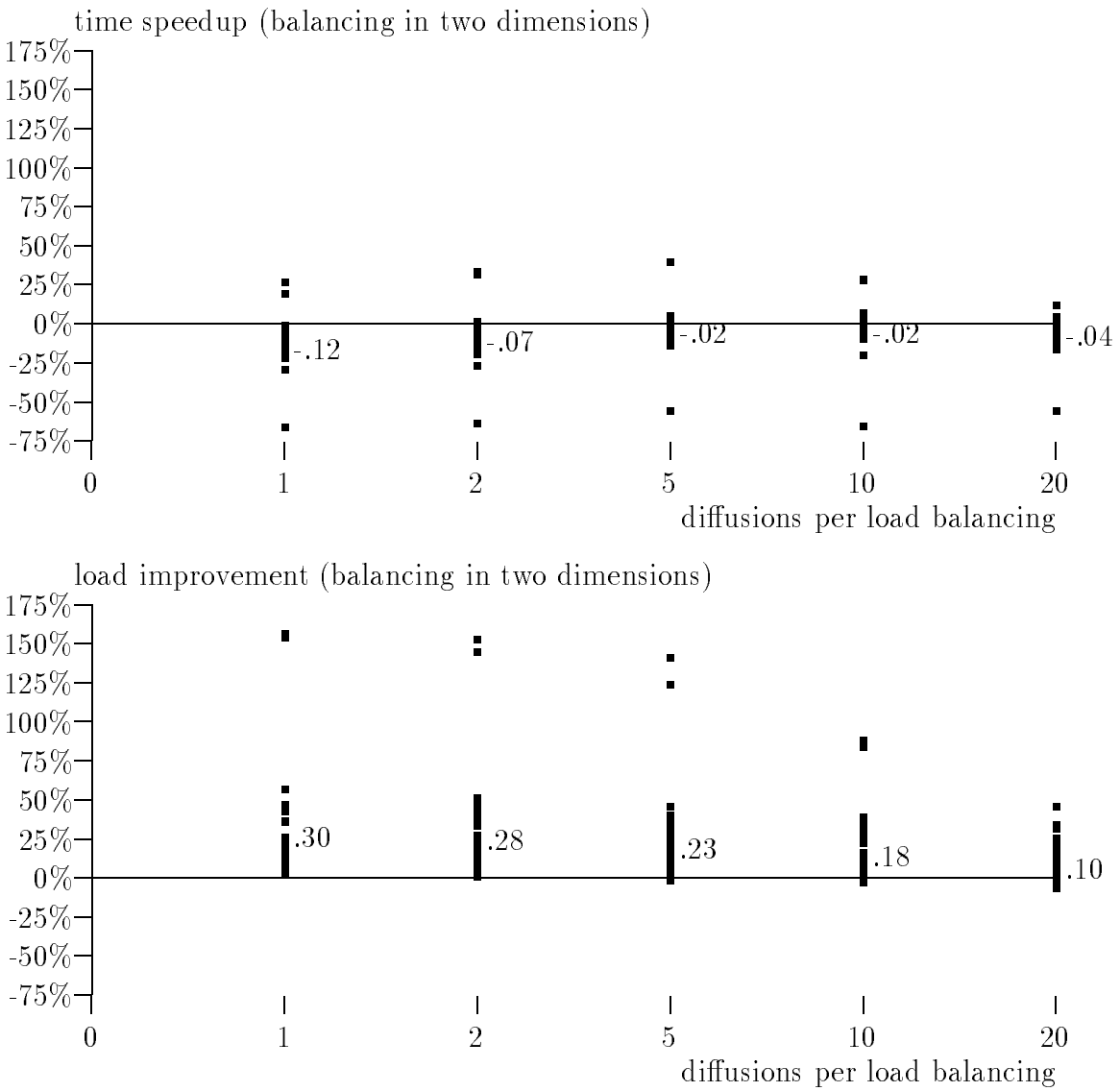

Figure 4.5: Average Speedup given by Scan Directed Load Balancing

of Scan Directed Load Balancing would yield useful speedup.

Figure 4.6 shows the results for the 18 images in the data set that are of size 512×512 pixels, corresponding to an average tile size of 8×8 pixels on each processor for UNC's model of the MasPar MP-1. A quick glance shows that, even though the improvement in load (compared to the average of all images) is mild, the improvement in time is sufficient to make the speedup positive for some frequencies of load balancing; this shows that in the larger images the performance improvements given by Scan Directed Load Balancing are sufficient to compensate for the overhead of load balancing. Larger images or problems with greater computational loads than Variable Conductance Diffusion can be expected to benefit even more from load balancing.

Another way of decreasing the impact of the load balancing overhead is to balance in only one dimension. Figure 4.7 shows the speedup given by balancing in either the row or the column dimension, or by balancing in alternate dimensions, and compares the speedup to that given by balancing in both dimensions on each load balancing cycle. This graph shows the average performance over all images, with load balancing done every 5 diffusion steps.

4.5.3 Optimal Performance

The performance discussed above gives the average over an indiscriminate sample of images that were not especially selected because of their load distribution. In particular, when Variable Conductance Diffusion is run on many of them, the load is almost even and most of the load balancing work is counter-productive.

Unlike these average images, it is easy to select images which only have a small area that needs to be diffused; for these images, load balancing can be very effective in optimizing the computation. This is true for regular diffusion, in which any pixel which matches its neighbors' average is inactive, and even more so for Variable Conductance Diffusion, in which both pixels which match their neighbors and pixels on or near sharp edges do not participate in the diffusion. Images which have been processed using Adaptive Histogram Equalization (AHE) [24] are an example of images which only have a small area that needs to be diffused. AHE removes most of the low-frequency noise from images, so what is left is either high-frequency noise (edges) or flat, evenly shaded surfaces. The image in file `bowahes8.im` is just such a file, with only one small area that is active during diffusion.

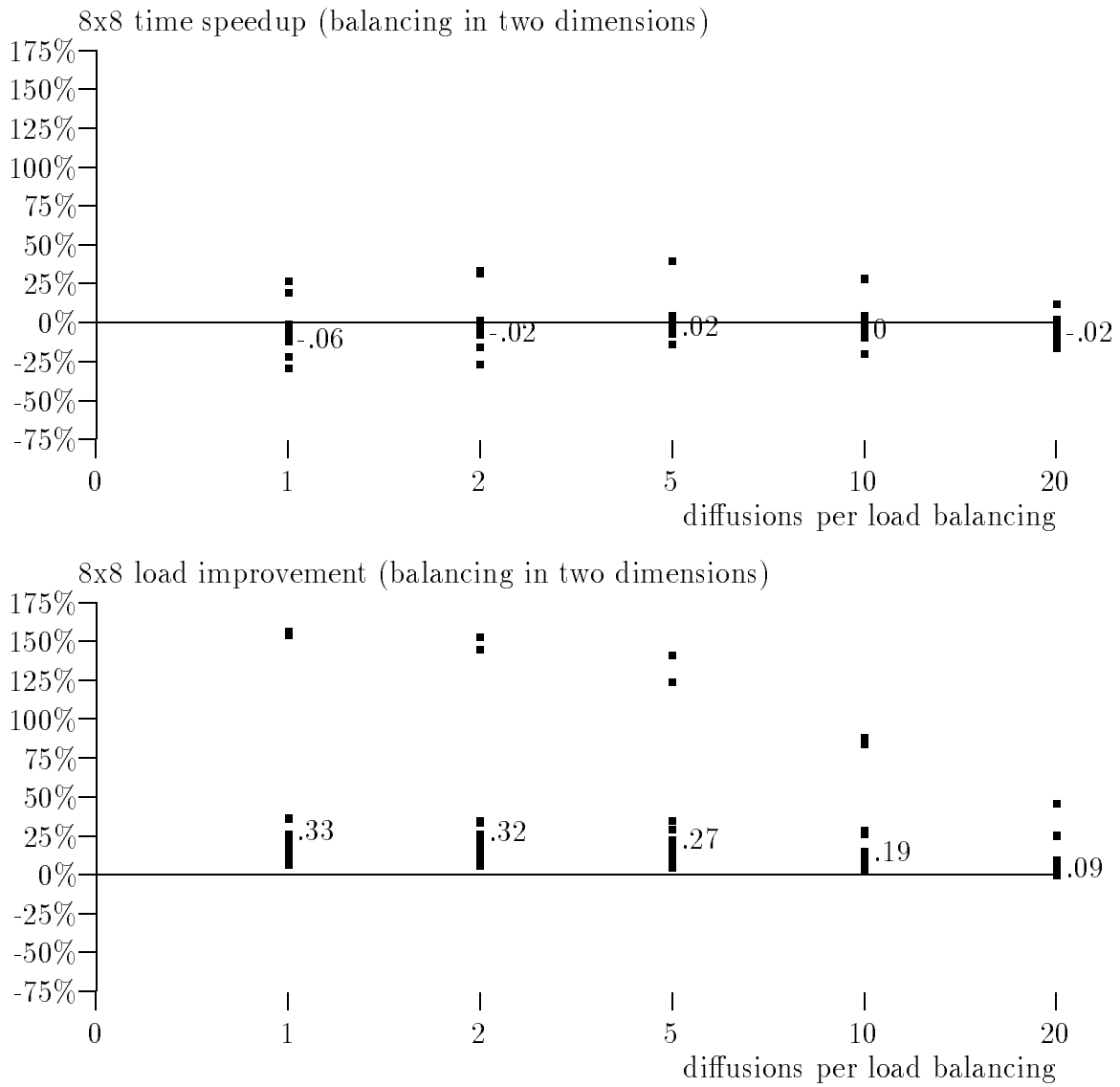

Figure 4.6: Average Speedup for 512×512 Images

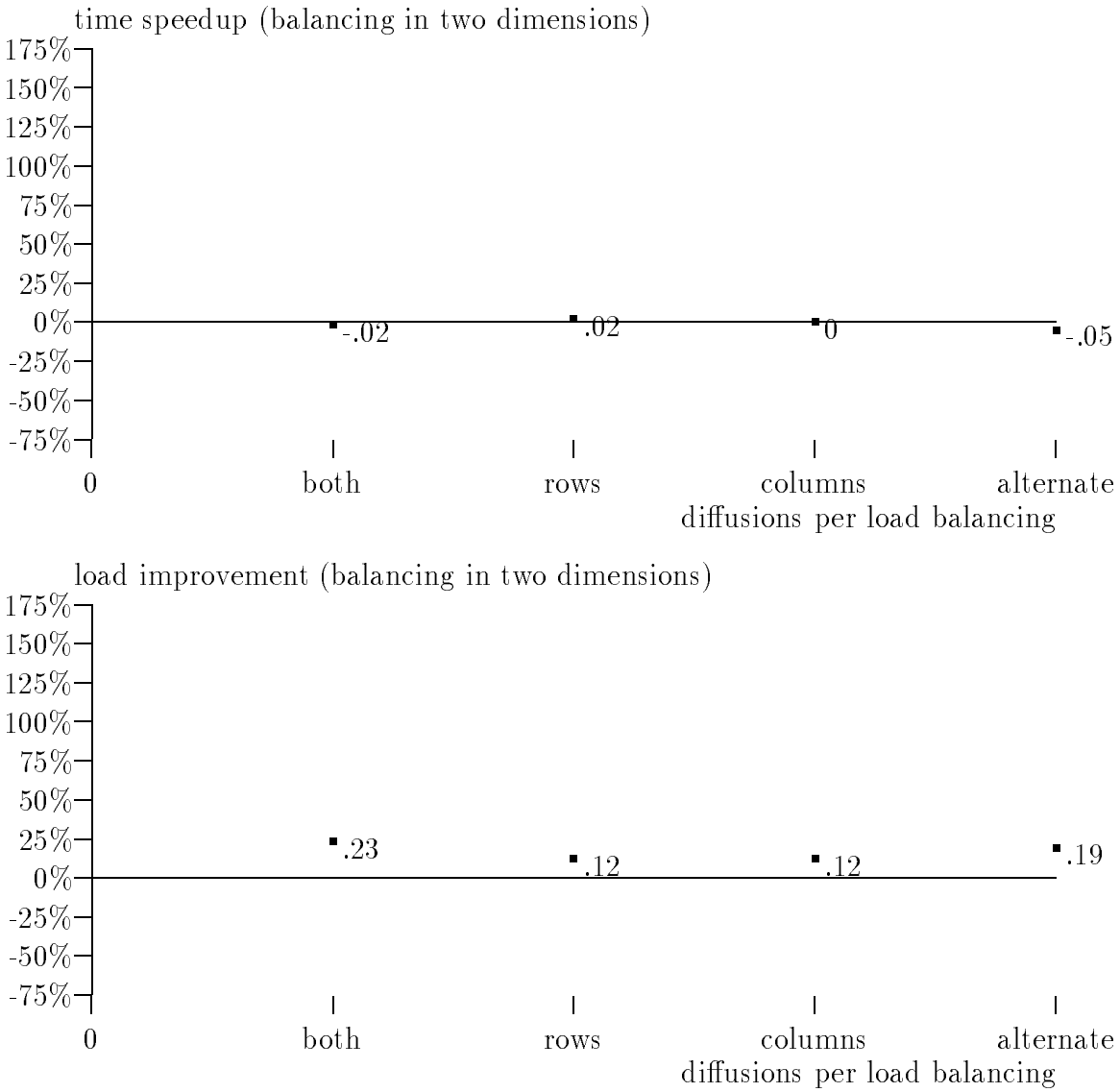

Figure 4.7: Average Speedup by Balancing in Different Dimensions

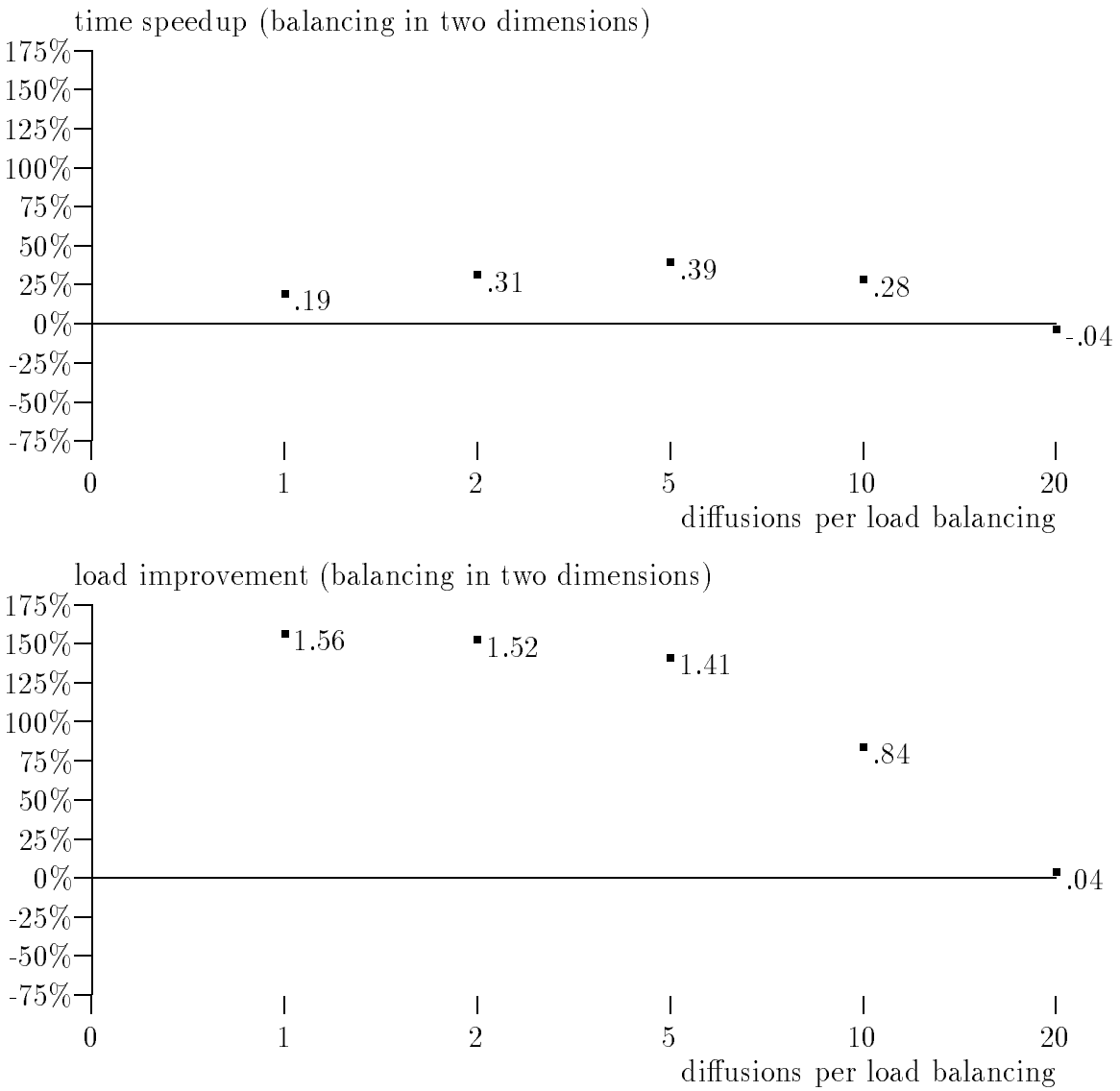

Figure 4.8: Speedup for image `bowahes8.im`

The speedup given by Scan Directed Load Balancing for this file is shown in Figure 4.8. The 150% improvement in the load of the most heavily loaded processor, which is achieved by load balancing after every diffusion iteration, only results in a 20% speedup of the algorithm as a whole, so the optimization process is clearly suffering from some inefficiencies. Part this inefficiency can be attributed to the cost of balancing the load, since reducing the frequency of load balancing increases the speedup, up to almost 40% when load balancing is performed once for every 5 diffusion iterations; the remainder can be attributed to the fact that inactive points do not actually have a load of zero, as assumed by this version of Scan Directed Load Balancing.

The slowdown that occurs when balancing is performed once every 20 diffusion iterations is due to the cost of balancing the load and other costs not being sufficiently compensated by the resulting speedup.

4.6 Analysis

4.6.1 Modeling the Load

In all of the above graphs, the time speedup of Scan Directed Load Balancing is less (and usually negative) when load balancing is most and least frequent; the speedup is greatest for some intermediate frequency of load balancing. This phenomenon can be explained by looking at both the costs and the benefits of frequent load balancing.

As load balancing is done more and more frequently, the benefits increase, but nonlinearly: load balancing makes a big difference if it is applied when the load is very uneven, but if the load is nearly even, load balancing improves a situation that is already near-optimal, and therefore is not as effective. That this is the case can be seen in Figures 4.5 and 4.6. In Figure 4.5, for instance, the load improvement given by doing load balancing twice (once after every 20 diffusion cycles in a 50 cycle run) is 10%. Load balancing 4 times (once every 10 cycles), only gives an additional improvement of 8%, balancing 9 times an additional improvement of 5%, and so on.

To improve on these figures, load balancing could be performed only after a substantial imbalance has been detected in the computation, so load balancing would not waste time trying to improve a load that is already almost balanced. Unfortunately, an imbalanced load is not necessarily one that can

be rebalanced within OMS constraints, so this approach is unlikely to effectively distinguish distributions that can be improved by load balancing from those which can't. A better approach is compute hyperplane loads, and only load balance if the hyperplane loads are mismatched. This is a promising approach, even though computing hyperplane loads takes a substantial fraction (on the MP-1) of the time required for balancing the load; exploring this option is one of the likely directions for future work.

There is little theoretical support for modeling the load improvement with any particular equation, since the load imbalance depends on the computation being balanced and on the particular data which the computation is processing; for example, the same Variable Conductance Diffusion computation provides very different load improvements for the average image, for images of size 512×512, and for the image `bowahes8.im`. However, all load improvement curves $ld = f(n)$, with ld the ratio between the balanced and the unbalanced load and n the number of times that the load is balanced, are monotonically increasing; moreover, the derivative becomes less and less as n increases. Curves with this property can be modeled by equations of the form

$$ld = a(1 - e^{-kn}) \tag{4.1}$$

for some value of the coefficients a and k. This equation has little *predictive* value, since it has no theoretical basis, but the values of the coefficients can usefully *summarize* the behavior of Scan Directed Load Balancing for a particular computation on particular inputs.

Equation 4.1 is far from accurate, but at least approximately models the behavior of the performance of Scan Directed Load Balancing for the cases studied. Values of $k \approx 0.25$ and $a = 2.6$ approximately match the data shown in Figure 4.8, whereas $k = 0.5$ and $a = 1.3$ approximately match the data in Figure 4.5. At a glance, this suggest that even though the load improvement given by load balancing every time ($n = 50$) is much greater for `bowahes8.im` than for the average image, since 2.6 is a much greater ratio than 1.3, the improvement for low values of n is not as spectacular, since $0.25 < 0.5$.

4.6.2 Time Required to Balance Load

n	Time	Load Improvement	Computation Time	Balancing Time	Balancing Time / n
0	35.7s	–	35.7s	–	–
2	37.0s	4%	34.5s	2.5s	0.7s
4	28.0s	84%	18.2s	9.8s	3.2s
9	25.8s	141%	14.8s	11.0s	1.7s
24	27.1s	152%	14.2s	12.9s	0.7s
49	30.1s	156%	13.9s	16.2s	0.4s

Table 4.1: Estimate of Load Balancing time for `bowahes8.im`

The load balanced Variable Conductance Diffusion program `bedd` outputs the maximum load per processor on each diffusion cycle and the average maximum load over all diffusion cycles. Taking this average maximum load to be proportional to the time actually spent computing useful load, and estimating all the remaining time to be spent on balancing the load, gives an estimate of the time required for load balancing. An example of this estimate for the time required to balance the load for the image in `bowahes8.im` is shown in Table 4.1. This estimate is probably an overestimate, since there are overhead costs of computing on a balanced load that are not present in an unbalanced Variable Conductance Diffusion, such as the fact that the maximum tile size increases and therefore more communication might be required for transferring the edge pixels; the column labeled "Balancing Time" above therefore includes both the time for balancing and the time for all parts of the computation that do not require less time as the load decreases. Using the data in Table 4.1, for instance, the total time required for load balancing and the overhead of data structure management varies between 400 and 3200 milliseconds per load balance.

Since the time tlb for load balancing and other overheads is monotonically but not linearly increasing, it can be modeled, as can the load and with the same caveats about lack of theoretical supports, by a function such as

$$tlb = r(1 - e^{-hn}) \tag{4.2}$$

A reasonable fit to the data of Table 4.1 is obtained for $r \approx 20$s and

$h \approx 0.1$.

For the case of $n = 49$, profiling shows that approximately 1/3 of the time is spent in the load balancing routines, and most of the rest of the time in the Variable Conductance Diffusion routine. Since the total run time is 30.1 seconds, approximately $30.1/3 \approx 10$ seconds are spent balancing; since the load is balanced 49 times, about 200 milliseconds are sufficient to balance the load once. The remaining 8.4 seconds of the "Balancing Time" in Table 4.1 are the time required to manage data structures in memory and to do fixed cost operations such as exchanging edges of tiles.

This time for load balancing compares well with the results of Dragon and Gustafson [28] for the NCUBE hypercube, a MIMD computer in which each processor is more considerably powerful than the MP-1's processors. Dragon and Gustafson have implemented a version of Orthogonal Recursive Bisection to load balance the NCUBE and have run it on a program that did little more than unbalance the load. The data averaged eight particles[3] per processor, and measured a load balancing time of 121 milliseconds on 1024 processors, their largest configuration. Using their formula, $t = 4.4564 - 4.0412D + 1.5261D^2$, to extrapolate to a hypercube of 4096 processors ($D = 12$) gives a time of about 175 milliseconds for load balancing alone. Considering that Variable Conductance