Chapter 3

The Algorithm

I am told the only real explanation for it is mathematical.
— John Cassidy, "The Aerobie Book"

3.1 Assumptions and Definition of Terms

The data D is a k dimensional grid in which every axis has length n, for a total of n^k data points. Each data point is indexed by a vector H of the form $H = [H_1, \ldots, H_{k-1}]$, with $0 \leq H_x < n$ for each dimension x. The computation is carried out by a k dimensional mesh-connected machine M with p^k processors; $p < n$ since for $p \geq n$ load balancing is not required. Each processor is indexed by a vectors I of the form $I = [I_1, \ldots, I_{k-1}]$, with $0 \leq I_x < p$. The variable x is used to index over the dimensions, so $0 \leq x < k$.

3.1.1 Domain Decomposition

A Domain Decomposition describes the distribution of data over the processors of a machine.

Definition 2 *A* domain decomposition μ *maps each data point index vector* H *of the data grid* D *to a processor index* I*:* $\mu(H) = I$.

The domain decompositions that are described and used in this chapter are *orthogonal*, *monotonic*, and *surjective* (OMS).

Definition 3 *A domain decomposition μ is* orthogonal *if it can be written as $\mu([H_0, \ldots, H_{k-1}]) = [\mu_0(H_0), \ldots, \mu_{k-1}(H_{k-1})]$, where $\mu_0, \ldots, \mu_{k-1}$ are functions with domain $\{0, 1, \ldots, n-1\}$ and range $\{0, 1, \ldots, p-1\}$.*

In other words, an orthogonal decomposition maps each dimension of the data grid independently, so that $I_x = \mu_x(H_x)$.

Definition 4 *An orthogonal domain decomposition μ is* monotonic *if each μ_x is monotonic, i.e., if for any H_x, H'_x*

$$H_x \leq H'_x \Rightarrow \mu_x(H_x) \leq \mu_x(H'_x)$$

In a monotonic orthogonal decomposition, each processor is assigned a contiguous, rectangular portion of the grid. Such a decomposition is uniquely determined by the $k \times (p-1)$ values giving the boundaries in each dimension of the distribution of data among processors. In two dimensions with four processor rows and four processor columns, for instance, there are three boundaries in each of the two dimensions; each boundary separates the data that is in one row or column from the data in the next row or column.

Definition 5 *A orthogonal monotonic domain decomposition μ is* surjective *if each I is the image of some H, i.e.*

$$\forall I \exists H : \mu_x(H) = I$$

The surjective property of OMS decompositions simply states that the subgrid assigned to each processor is nonempty.

In each dimension x, a processor I holds $\eta_x(I)$ grid points, $0 < \eta_x(I)$ for all x and I, for a total of $\Pi_x \eta_x(I)$ grid points on processor I. These grid points are indexed by a local index vector G of the form $G = [G_0, \ldots, G_{k-1}]$, with $0 \leq G_x < \eta_x(I)$. Since the domain decomposition is orthogonal, the range for G_x is the same for all processors with the same index I_x. The largest value of $\eta_x(I_x)$ in dimension x is $\eta_x = \max_i(\eta_x(I))$, where $I = [I_0, \ldots, I_x = i, \ldots, I_{k-1}]$.

The *uniform hierarchical decomposition* $\hat{\mu}$ partitions the data grid D into k dimensional subcubes in which the sides of every subcube have length either $\lfloor n/p \rfloor$ or $\lceil n/p \rceil$.

Definition 6 $\hat{\mu}(H) = [\hat{\mu}_0(H_0), \ldots, \hat{\mu}_{k-1}(H_{k-1})]$, *and*

$$\hat{\mu}_x(H_x) = \lfloor H_x \frac{p}{n} \rfloor$$

The definition makes it immediately evident that $\hat{\mu}$ is orthogonal. Since $0 < p < n$, $p/n > 0$ so $\hat{\mu}$ is monotonic; since $p/n < 1$, for each I_x there exists at least one H_x such that $\lfloor H_x p/n \rfloor = I_x$, so $\hat{\mu}$ is also surjective.

Scan Directed Load Balancing is started with the uniform hierarchical decomposition $\mu = \hat{\mu}$ and varies it in response to the measured distribution of active data. Repeated applications of Scan Directed Load Balancing change μ but preserve its monotonic, orthogonal, and surjective properties.

3.1.2 Distance and Neighborhood

Definition 7 *The* distance $\delta(I, I')$ *between two processors with indices* I *and* I' *in a rectangular mesh is given by the Manhattan metric*

$$\delta(I, I') = \sum_x |I_x - I'_x|$$

The distance $\delta(H, H')$ *between two data points with indices* H *and* H' *in a rectangular grid is likewise*

$$\delta(H, H') = \sum_x |H_x - H'_x|$$

The distance between two processors in the mesh is the number of links in the shortest path between the two processors. The analogous definition for distance between two data points is important for local algorithms which update the value at one point in a grid by only accessing data within a specified radius of the point. Of particular interest are data points which are a distance 1 from each other.

Definition 8 *The* neighbors $\nu(P_I)$ *of a processor* P_I *are the* $2k$ *processors at distance* 1 *from* P_I*:*

$$\nu(P_I) = \{P_{I'} : \delta(I, I') = 1\}$$

Similarly, the neighbors $\nu(D_H)$ of a data point D_H are the $2k$ points $D_{H'}$ at distance 1 from D_H:

$$\nu(D_H) = \{D_{H'} : \delta(H, H') = 1\}$$

All processors and data points have $2k$ neighbors except for those at the *edge* of a mesh or grid, which may have as few as k.

Table 3.1 gives a condensed version of the above definitions and of other symbols used below.

3.1.3 Locality Property

Definition 9 *A function on a grid is* local *if at each point it computes a function of only the point and its neighbors.*

Definition 10 *A domain decomposition μ is* locality preserving *if*

$$\mu(\nu(H)) \subseteq \nu(\mu(H))$$

Another way of expressing this is that μ is locality preserving if the mesh distance between any two neighboring points is either 0 or 1.

An implementation of a local operation or a local function using a locality preserving decomposition only requires communication between neighboring processors. OMS decomposition always store data points in $\nu(D_H)$ either on processor $I = \mu(H)$ or on one of the processors in $\nu(P_I)$, as shown by Theorem 1, and therefore preserve locality.

Domain decompositions which preserve locality are said to have the *Locality Property.*

The locality property can be extended to neighborhoods ν_s of fixed size $s > 1$. Such extensions can be very useful in practice: recent work by Bromley et al. [18], for instance, has produced a specialized compiler that optimizes arbitrary computations over local templates of fixed, but unlimited size. If an algorithm requires information exchange among grid points which are no more than distance s apart, the data can be mapped with an OMS decomposition and points within grid distance s are no more than s processors apart. Alternately, if s is small compared to n/p, the average number of points per processor in each dimension, the surjective requirement of OMS decompositions can be modified so each processor has at least s points in

Symbol	Value or Definition	Explanation
k	$integer$	dimension of the mesh and data
x	$0, \ldots, k-1$	index over dimensions
p^k	$p \gg 1$	number of processors
I	$[I_0, \ldots, I_{k-1}]$	processor index (a vector)
I_m	$[p-1, \ldots, p-1]$	maximum processor index
P_I		processor at index I
$P_x(i)$		processor hyperplane P_I where $I_x = i$
A_I		active points on processor P_I
A_{tot}	$\sum_I A_I$	number of active points
n^k		number of data points
H	$[H_0, \ldots, H_{k-1}]$	data point index (vector)
D_H		data point at index H
$\mu(H)$	$\mu : H \rightarrow I$	mapping of data points to processors
$\hat{\mu}(H)$		uniform hierarchical decomposition
b^k	$b > n/p$	number of buffers per processor
η_x	$1, \ldots, b$	max data index in dimension x
$\eta_x(I_x)$	$1, \ldots, b$	max data index on x-hyperplane i
G	$[G_0, \ldots, G_{k-1}]$	local (per-processor) data point index
$D_x(i, g)$		data hyperplane at index g in $P_x(i)$
$\delta(I, I')$	$\sum_x \lvert I_x - {I'}_x \rvert$	Manhattan distance I to I'
$\nu(X_I)$	$\{X_{I'} : \delta(I, I') = 1\}$	neighbors of processor or data point X_I
$\bar{A}$	$\max_x(A)/p$	average load per processor hyperplane
$\sigma_i(V)$	$\sum_{j<i} V_j$	left-to-right plus-scan of V
ℓ_x		max. load of any data x hyperplane
L_i	$\sigma_i(\max_x(A))$	activity over hyperplanes $< i$
F_i	$\bar{A} * i - L_i$	active point flow, i to $i-1$
f_i	$\lfloor F_i \rfloor$ or $\lceil F_i \rceil$	points actually shifted from i to $i-1$
$\rho(x, \oplus, A)$	(Definition 17)	$\oplus$ multi reduction across axis x of A
$W_x(i, g)$	(Definition 15)	activity of data x-hyperplane (i, g)
$S_{x,i}$	$\sum_{g<\eta_x(i)} W_x i, g$	activity of processor x-hyperplane i
α	$\bar{A}/(b - n/p)$	load equivalent of inactive data point

Table 3.1: Notation

each dimension. This modified version of OMS insures that points within grid distance s of each other are no farther than the neighboring processors.

Because the extension to larger neighborhoods is predictable, in what follows only neighborhoods of size $s = 1$ are considered explicitly and locality is defined strictly so that only communication between neighboring processors is local.

Definition 11 *An iterated* Local Computation *is a computation of the form* $D_H^{t+1} = f(D_H^t, \nu(D_H^t))$ *for some local function* f*, where* t *identifies successive iterations of the computation.*

Even though this definition only covers iterative computations, most computations that require high performance are iterative, so in practice the definition is not limiting. Examples of local computations are given in Section 2.3. All local computations are continuum computations, but the converse is not true.

Local computations can benefit from a careful domain decomposition, and in particular many are programmed to use the uniform hierarchical decomposition. If in a local computation $D_H^{t+1} = D_H^t$ for some, but not all values of H, then the function f need not be computed explicitly at H and the load of the data point D_H is 0; otherwise, the load is 1. This load may be unevenly distributed, according to the distribution of data points with load 0 and load 1. If the distribution is uneven, load balancing must change the domain decomposition μ so the load is spread more evenly. As long as μ remains orthogonal, monotonic, and surjective, the locality property holds and communication that is local under $\hat{\mu}$ is still local under the new μ produced by load balancing.

3.1.4 The Locality Property of OMS Decompositions

The conditions on μ given in Figure 3.1 are preserved by Scan Directed Load Balancing, and are sufficient to ensure that the decomposition has the Locality Property. In other words, any Orthogonal, Monotonic, and Surjective (OMS) decomposition has the locality property.

Theorem 1 *If the decomposition is OMS, the Locality Property holds.*

A decomposition μ is OMS if and only if for any data indices H and H', for any processor index I, and for any axis x,

$$H_x = H'_x \Rightarrow (\mu(H))_x = (\mu(H'))_x \quad (\mu \text{ orthogonal})$$
$$H_x \leq H'_x \Rightarrow (\mu(H))_x \leq (\mu(H'))_x \quad (\mu \text{ monotonic})$$
$$\exists H : \mu(H) = I \quad (\mu \text{ surjective})$$

Figure 3.1: Orthogonal, Monotonic, Surjective Decomposition μ.

Proof Assume that the theorem is not valid, so the Locality Property does not hold for some OMS decomposition μ. Take any point $H' \in \nu(H)$ such that $\mu(H') \notin \nu(\mu(H))$. Since $H' \in \nu(H)$ if and only if $\delta(H, H') \leq 1$ and $\mu(H') \notin \nu(\mu(H))$ if and only if $\delta(\mu(H), \mu(H')) > 1$, and given $I = \mu(H)$ and $I' = \mu(H)'$, either **(1)** I and I' differ in at least two dimensions x and y, $x \neq y$, or **(2)** I and I' differ only in dimension x, so that $\forall y \neq x, (I_y = I'_y)$.

If I and I' differ in at least two dimensions x and y, by Definition 7 $\delta(H, H') = 1$ only if H and H' differ in at most one dimension, z, such that either $x = z$ or $y = z$ but not both (since $x \neq y$). If $x = z$, then $H_x = H'_x$ implies μ is not orthogonal, since $\mu_x(H) \neq \mu_x(H')$; likewise if $y = z$.

Conversely if I and I' differ only in dimension x, assume without loss of generality that $I_x < I'_x$. Since $\delta(I, I') > 1$, $I'_x - I_x > 1$, so there is some processor I'', such that $I_x < I''_x < I'_x$ and $\forall y \neq x, (I_y = I''_y)$. Since μ is surjective, there is at least one data point $D_{H''}$ such that $I'' = \mu(H'')$. Then, since μ is monotonic, $H_x < H''_x < H'_x$, so $\delta(H, H') > 1$, which contradicts the claim that $\delta(H, H') = 1$.

□

Corollary *The Locality Property is preserved by any algorithm that preserves the OMS property.*

3.2 One Dimensional Load Balancing

3.2.1 Assumptions

This section describes Scan Directed Load Balancing for cases in which $k = 1$, so the computer is a linear array of processors and the data is a linear array

of data points. Examples for the Load Computation Step and the Data Movement Step of the Load Balancing Algorithm are given in Figures 3.3 and 3.4, respectively, and only show active data points.

In one dimension, the processor index I is a vector of length one and is represented by the scalar $i = 0, 1, \ldots$. The load A_i on each processor P_i is defined to be the number of active data points stored on processor P_i; the goal of load balancing is to make the maximum value of A_i as low as possible. The symbols used in this section are described in Table 3.2.

Symbol	Type	Explanation
i	int	**index** over processors: $0 \leq i < p$
A_i	int	number of **active** data points in processor P_i
L_i	int	number of active points to the **left** of P_i, $\sum_{j<i} A_j$
$\bar{A}$	rational	average number of active data points per processor
F_i	rational	active point **flow** P_i to P_{i-1} for balancing the load
f_i	int	actual leftward **flow** of active data points from P_i
$D_{i,0}$	data point	leftmost data point on processor i, if any
active	predicate	is the given data point active?

Table 3.2: Notation for One Dimension

This section assumes that each active data point has a load of 1 and each inactive data point has a load of 0. This assumption is relaxed in the next section, where each data hyperplane may have a different load, but used in this section to simplify the exposition.

3.2.2 Scans

Definition 12 *A* $scan(\oplus, V)$ *on a vector* $V = [V_0, V_1, \ldots, V_{n-1}]$ *with associative operator* $\oplus$ *yields the vector of partial results* $[id, V_0, V_0 \oplus V_1, \ldots, V_0 \oplus V_1 \oplus \ldots \oplus V_{n-2}]$, *where id is the identity value for* $\oplus$.

The identity value is 0 for $\oplus = +$ and is $-\infty$ for $\oplus = \max$.

This definition of scan is slightly different from the APL definition of scan, which returns values $[V_0, V_0 \oplus V_1, \ldots, V_0 \oplus V_1 \oplus \ldots \oplus V_{n-1}]$; the APL value can be obtained from the value computed by this scan by the local vector operation $scan(V)_i \oplus V_i$.

Definition 13 *A* plus-scan *or scan-add* $\sigma(V) = scan(+, V)$ *is a scan on an integer vector* V *with the operator* $+$ *and the identity value* 0.

The plus-scan is the only scan required for Scan Directed Load Balancing. Other scans are required for specific applications of Scan Directed Load Balancing, most notably the SOP interpreter described in Chapter 5.

```
1. Load Computation Step
   forall (i) in parallel
     L_i ← Σ_{j<i}(A_j)                    activity to left of processor i
     Ā ← (L_{p-1} + A_{p-1})/p             Ā broadcast to all processors
     if (Ā < 1) then abort endif           avoid underflows (see below)
     F_i ← iĀ − L_i                        flow from processor i to i − 1
   endfor
2. Data Movement Step
   forall (i) in parallel
     while (F_i > 1/2)                          tested by each processor
       if active(D_{i,0}) then F_i ← F_i − 1 endif
       put ()
     endwhile
     while (F_i ≤ −1/2)
       get ()
       if active(D_{i,0}) then F_i ← F_i + 1 endif
     endwhile
   endfor
```

Underflows and their consequences are described in detail in Section 3.4.3.

Figure 3.2: One-Dimensional Scan Directed Load Balancing

3.2.3 Load Computation Step

The code for one dimensional Scan Directed Load Balancing has two steps, shown in Figure 3.2. The first step does **Load Computation**, by computing the data movement required to transform the old mapping μ^t to the new mapping μ^{t+1}. A scan of the processor loads A_i gives the load L_i to the left of each processor. The ideal load is the average load $\bar{A}$, computed as the total

load ($L_{p-1} + A_{p-1}$) divided by the number p of processors. The difference between the actual load to the left and the desired load to the left, $i\bar{A}$, gives the *flow* F_i, the number of active points to be transferred from processor P_i to processor P_{i-1} to obtain a perfectly balanced load. $F_i < 0$ means $|F_i|$ points are transferred from processor P_{i-1} to P_i.

The Load Computation Step restated in vector terms computes a vector F of flows given a vector of processor loads A. The vector F is then used in the Data Movement step, while the intermediate values L (a vector) and $\bar{A}$ (a scalar) are no longer used. The Load Computation Step expressed in APL is:

$$F \leftarrow ((\iota p) \times (+/A) \div p) - +\backslash A$$

Since the average load $\bar{A}$ is a rational number, the flow F_i is likewise rational. This may cause some confusion as to exactly what must be shifted if, for instance, the flow is 3/4. In this algorithm the flow is always rounded down, for reasons explained in Section 3.5, so the actual flow f_i is $f_i = \lceil F_i - 0.5 \rceil$. A value of $F_i = 0.5$ is therefore rounded down to $f_i = 0$, whereas $F_i = 3/4$ is rounded up to $f_i = 1$.

active data points	a	bcd	ef	–	gh	i	–	jkl
processor index i	0	1	2	3	4	5	6	7
A_i	1	3	2	0	2	1	0	3
L_i	0	1	4	6	6	8	9	9
$\bar{A}$	1.5	1.5	1.5	1.5	1.5	1.5	1.5	1.5
F_i	0	0.5	-1	-1.5	0	-0.5	0	1.5

Figure 3.3: Example of values produced during the Load Computation Step.

The example in Figure 3.3 shows the Load Computation Step on an 8-processor linear array with 12 active data points labeled a through l. Processor P_1, for instance, has the 3 active points b, c, and d, so $A_1 = 3$. Processor P_1 also computes $L_1 = 1$, since there is 1 active data point (a in this case) to its left. The average $\bar{A} = 12/8 = 1.5$ in this case is not an integer, so the final load in each processor is going to be either $\lfloor 1.5 \rfloor = 1$ or $\lceil 1.5 \rceil = 2$. Since $L_1 = 1$ and $\bar{A} = 1.5$, the net flow for balancing the load in P_1 is $F_1 = 0.5$.

The actual flow is going to be $f_1 = \lceil F_1 - 0.5 \rceil = \lceil 0 \rceil = 0$, and no points will be shifted from processor P_1 to processor P_0. By comparison, $F_2 = -1$ so $f_2 = -1$ and one active data point will be shifted in the Data Movement Step from P_1 to P_2. Note that f_i, like μ, is never computed explicitly.

The Load Computation Step must check that the average load is at least one; if it were less than one, adjacent active data points would be distributed over more than two processors, causing μ to not be surjective. The situation where the average load is less than one is known as *underflow* and is discussed in Section 3.4.3.

3.2.4 Data Movement Step

The second step in the algorithm does data **Movement**, shifting the data from one processor to another until each data point D_H is in processor $P_{\mu(H)}$. The data movement uses the subroutines `put ()` and `get ()`, which put the leftmost data point of P_i onto the rightmost end of the data buffer of P_{i-1}, or get the rightmost data point from P_{i-1} and store it as the leftmost point in P_i, respectively. The function of `put ()` and `get ()` is to maintain one double-ended queue in each processor, such that one end of each queue communicates with one of the processor's two neighbors. The queue discipline insures that no data point ever gets out of order, which is essential for guaranteeing that the locality constraints hold.

The data movement step itself has two phases. First, processors with $F_i > 0$ shift points to the left until $f_i = \lceil F_i - 0.5 \rceil$ active points have been shifted. Then, processors with $F_i < 0$ take points from the left until $-f_i$ active data points have been shifted.

The predicate `active()` in Figure 3.2 is used to distinguish active points from inactive points. Inactive points, described in Section 2.3.4 and further analyzed below, do not contribute to the load and are therefore irrelevant to load balancing. The value of F_i is only updated if the data point that was shifted is active, so inactive data points are handled transparently.

The implementation of `active()` is not important, as long as it correctly returns `true` if the data point is active, and `false` if there is no such point or the point is inactive. One possible implementation is to keep a vector of booleans to record which data points are active. Another implementation is described in Section 3.4.4.

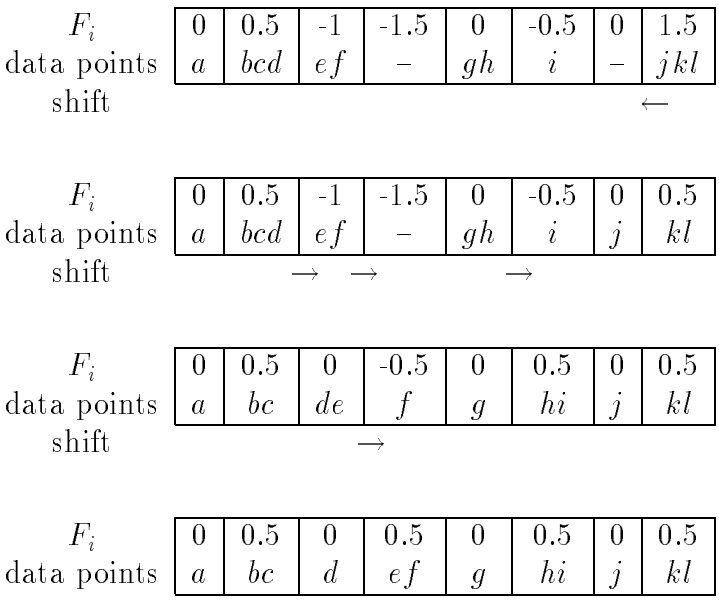


Figure 3.4: Load Transfer in the Data Movement Step.

The example in Figure 3.4 shows the Data Movement Step for the distribution of active data points given in Figure 3.3. In this example there are no inactive points, so all 12 data points are active. The first loop iterates once, since the maximum value of f_i is 1; the second loop iterates twice, since the minimum value of f_i is -2. At the end of the second loop all flows are $0.5 < F_i \leq 0.5$, and the load on each processor is $\lfloor \bar{A} \rfloor \leq A_i \leq \lceil \bar{A} \rceil$. Though the algorithm only shows processor i computing F_i, processor i can also compute $F_{i+1} = F_i + A_i$ and use the value to synchronize with its right neighbor, if any.

The optimality of the load balance that this algorithm achieves is discussed, with other results, in Section 3.5.

3.3 Multi Dimensional Load Balancing

3.3.1 Assumptions

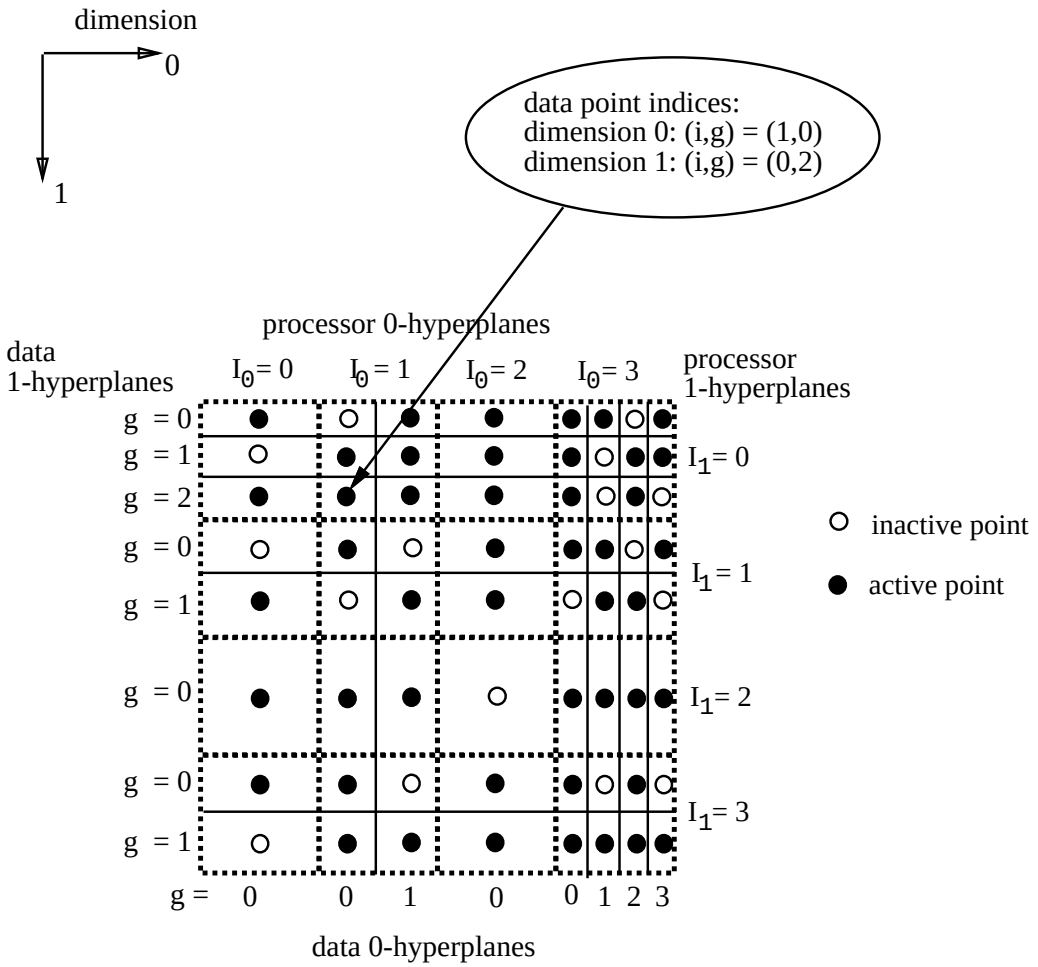
This section describes Scan Directed Load Balancing for cases in which $k > 1$, so the computer is a k dimensional mesh of processors and the data is a k dimensional grid of data points. For $k > 1$, the processor index I is an element of $\{0, \ldots, p-1\}^k$. The notation for multi dimensional Scan Directed Load Balancing is shown in Table 3.3.

In k dimensions, a data x-hyperplane is a $k-1$ dimensional array of data points, all with the same x-index. The dimension x which defines the hyperplane is *orthogonal* to the hyperplane. For example if $k = 2$ and dimension $x = 0$ is horizontal and dimension $x = 1$ is vertical, a 0-hyperplane is a column and a 1-hyperplane is a row. Data and processor hyperplanes are shown in Figure 3.5, where each dot represents one data point. In one dimension, each processor is a processor hyperplane, and each position of the data grid is a data hyperplane, whether or not there is a data point at that position.

If $k = 3$ and given that the dimensions are indexed $x = 0$ for the horizontal, $x = 1$ for the vertical, and $x = 2$ for the depth dimension: a 0-hyperplane is a vertical plane from the near to the far face, a 1-hyperplane is a horizontal plane, and a 2-hyperplane is the vertical plane orthogonal to the other two. Figure 3.6 shows all the processor hyperplanes in a $2 \times 2 \times 2$ cube of processors, and also an outline of one of the data 0-hyperplanes. Hyperplanes for

Symbol	Type	Explanation
k	int	number of dimensions
x	int	dimension along which load is being balanced
I	int vector	**index** over processors
i	int	processor index along dimension x
g	int	index **within processors** along dimension x
η_x	int	maximum data x-hyperplanes per processor
$\eta_x(i)$	int	number of data hyperplanes with index i
A_I	int	number of **active** data points in P_I
$A_{I,x,g}$	int	**active** points in P_I with x-index g
$W_x(i,g)$	int	load of data hyperplane at coordinates (i,g)
ℓ_x	int	maximum $W_x(i,g)$ over all i and g
S_i	int	load of processor hyperplane at coordinate i
L_i	int	number of active points to 'left', $\sum_{j<i} A_I : I_x = j$
$\bar{A}$	rational	average active points per processor hyperplane
F_i	rational	computed **flow** of activity P_i to P_{i-1}

Table 3.3: Notation for Multiple Dimensions



For this example, p = 4, n = 8
Not shown is b = 8 > n/p = 2

..... boundaries between processors
_____ boundaries between data planes

Figure 3.5: Explanation of Terms for $k = 2$

$k > 3$ are harder to visualize, but mathematically no more complex.

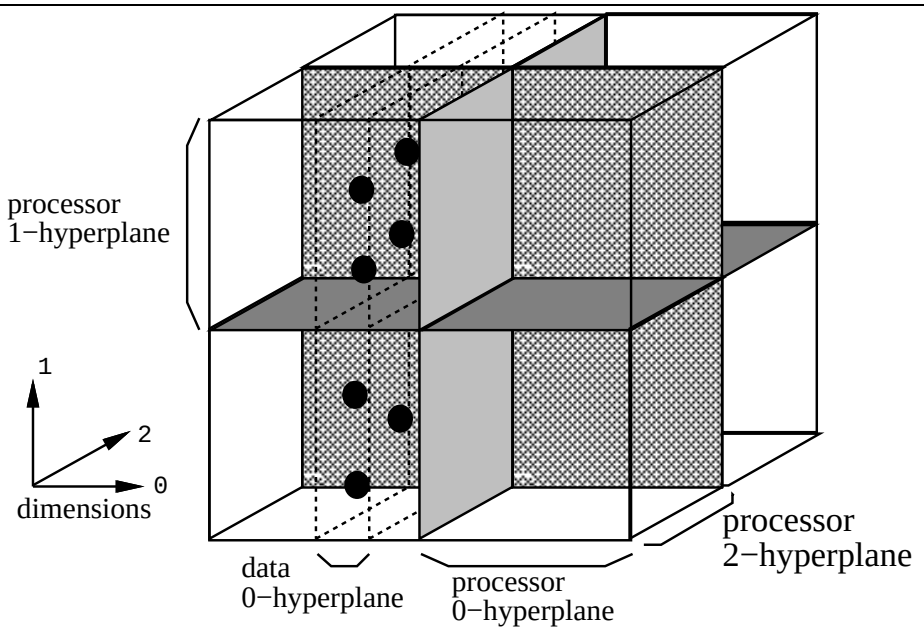


Figure 3.6: Data and Processor hyperplanes

Scan Directed Load Balancing for more than one dimension is similar to one dimensional Scan Directed Load Balancing, in that load balancing is done in each dimension using a generalization of the one dimensional algorithm. That algorithm was presented as balancing the load by shifting data points among processors; in generalizing to dimension $k > 1$, each of these data points is replaced by a hyperplane in k dimensional space. Balancing is achieved by shifting data hyperplanes among processor hyperplanes. For $k = 1$, hyperplanes are indistinguishable from data points or individual processors, and the two algorithms are identical.

Since Scan Directed Load Balancing only balances one dimension x at a time, the index i is used instead of the notation I_x to enumerate processor hyperplanes. Data hyperplanes are indexed within processors by $g \in \{0, 1, \ldots \eta_x(i) - 1\}$, so given an index x, a data hyperplane is uniquely identified by the pair (i, g), where i is a processor index along the x axis.

The load of each data x hyperplane is the *maximal* load illustrated in Figure 2.2, the maximum of the per-processor load of a hyperplane. The

k dimensional version of Scan Directed Load Balancing has a step that computes a one dimensional vector of loads of data hyperplanes; it is these loads which are balanced across the linear array of processor hyperplanes.

Definition 14 *A* per-processor hyperplane load $A_{I,x,g}$ *is the number of active data points stored in processor* P_I *that are within* x*-hyperplane* (i, g)*:*

$$A_{I,x,g} = |\{H : \mu(H) = I \wedge \text{active}(D_H)\}|$$

where $H = [H_0, \ldots, H_{n-1}]$ *and*

$$H_x = \sum_{j=0}^{I_x} \eta_x(j) + g$$

This value is computed by each processor for each hyperplane $(I_x, g) = (i, g)$ that intersects it, and used in computing the hyperplane loads $W_x(i, g)$.

Definition 15 *A* maximal hyperplane load $W_x(i, g)$ *for the* x *hyperplane of data with coordinates* (i, g) *is the maximum of its per-processor hyperplane loads.*

$$W_x(i, g) = \max(\{A_{I,x,g} : I_x = i\})$$

For example in Figure 3.5, $A_{[1,0],0,1} = 3$, since there are three active points in the processor with $I_0 = 1$, $I_1 = 0$ on hyperplane at $g = 1$ perpendicular to $x = 0$; likewise, $A_{[3,1],1,1} = 2$ since there are two active points with in the row $g = 1$ in the processor on column 3, row 1.

The same example has $W_0(1, 1) = 3$, since the column (hyperplane of data perpendicular to dimension 0) with coordinates $(i = 1, g = 1)$ (the third data column from the left) has a maximal hyperplane load of 3, since the topmost processor has three active points belonging to that column. 1-hyperplane (data row) at $(i = 3, g = 0)$ has a maximal hyperplane load of $W_1(i = 3, g = 0) = 2$, since the rightmost processor, which has the most active points of any processor for the second data row from the bottom, has two active points.

Taking as an example the configuration shown in Figure 3.6, assume the distribution μ is such that the highlighted data hyperplane has index $g = 2$

and has 4 active data points in its upper front section, as shown in the figure. The processor index I for the front upper left processor is 0 for dimensions 0 and 2, and 1 for dimension 1, so $I = [0, 1, 0]$. The load for the part of data 0 hyperplane $(0, 2)$ in processor $P_{[0,1,0]}$ is $A_{[0,1,0],0,2} = 4$. Also shown in Figure 3.6 is the fact that $A_{[0,0,0],0,2} = 3$.

As in the one dimensional case, multi dimensional Scan Directed Load Balancing has a Load Computation Step followed by a Data Movement Step; these two steps together constitute one iteration of the algorithm, which has a total of k iterations for load balancing in k dimensions.

3.3.2 Reductions

Definition 16 *A reduction$(\oplus, V)$ on a vector $V = [V_0, V_1, \ldots, V_{n-1}]$ with associative binary operator $\oplus$ yields the single result $V_0 \oplus V_1 \oplus \ldots \oplus V_{n-2}$.*

The word reduction is used in this dissertation with two similar but distinct meanings. One of the meanings of reduction is the computation of a single value from a vector or an array, and is taken from APL; the word suggests the change in size from a vector to a scalar. It is this meaning of reduction that is defined above. Another standard usage of the word reduction is in expressions such as graph reduction or string reduction, or the reduction of a functional language. This meaning of the word refers to a strategy or algorithm for executing a functional program, and to the implementation of the strategy. The two interpretations of the word share the meaning of applying an operation to a structured input to produce a "simpler" result: in the case of functional reduction, the operator is the interpreter for the functional language, the structured input is the expression given by a function with its arguments, and the "simpler" result is the value computed by the function. In current technical usage, however, the two meanings are quite distinct and not interchangeable.

Definition 17 *Let S be a set on whose elements is defined an associative operator oplus. Let Q be a collection of n^k elements of S indexed by J, where $J = [J_0, \ldots, J_{k-1}]$ with $0 \leq J_x < n$ for all x, $0 \leq x < k$. For all $0 \leq i < n$, let $\rho_i^x(Q) = \oplus\{Q_J : J_x = i\}$ be the result of applying $\oplus$ to all elements of Q whose index in dimension x is i. A* multi reduction *$\rho(x, \oplus, Q)$ in dimension x computes the vector*

$$\rho(x, \oplus, Q) = [\rho_0^x(Q), \ldots, \rho_{n-1}^x(Q)]$$

An example of multi reduction is given in Figure 3.7. The *multi reduction* with the operator max, which returns the largest of its two operands, is the only multi reduction used in Scan Directed Load Balancing. In a multi reduction all of the data with the same x index $H_x = h_0$ contributes a single value to the final result. The max multi reduction produces as its result the maximum value of each data hyperplane, a data x-hyperplane being the collection of all the data with the same index in the x dimension.

A single multi reduction can be used to simultaneously compute one hyperplane load on each hyperplane of processors: $W_x(*, g) = \rho(x, \max, A_{*,x,g})$ (the $*$ represents all values of $i \in \{0, \ldots, p-1\}$) computes the hyperplane loads for a single value of the local index g on all processor hyperplanes simultaneously.

A load heuristic different from the maximal load could be computed using an operator different from max in the multi reduction. Section 2.1.5 compares the maximal load heuristic to an additive load heuristic, where the operator $+$ is used instead of max.

3.3.3 Load Computation Step

Multidimensional load balancing uses the data hyperplane loads $W_x(i, g)$ just as one dimensional load balancing uses individual data point loads. The value $S_i = \sum_g W_x(i, g)$ takes the place of A_i in the one dimensional algorithm. The value $\eta_x(i)$ keeps track of how many data hyperplanes are stored in processor hyperplane i, so that the data hyperplane index g is always $g \in \{0, \ldots \eta_x(i) - 1\}$. In Figure 3.5, for example, dimension 0 has $\eta_0 = 4$ and $\eta_0(0) = \eta_0(2) = 1$, $\eta_0(1) = 2$, and $\eta_0(3) = 4$. Dimension 1 has $\eta_1 = 3$ and $\eta_1(0) = 3$, $\eta_1(1) = \eta_1(3) = 2$, and $\eta_1(2) = 1$.

As in the one dimensional case, a scan gives a vector L of loads on all processor hyperplanes $j < i$, and that vector is used to compute a vector of flows F.

The code for multi dimensional Scan Directed Load Balancing is shown in Figure 3.8.

The quantity Q for this example is the $3 \times 3 \times 2$ array with dimension $k = 3$.

$$\begin{array}{cccccc} 1 & 2 & 3 & 10 & 11 & 12 \\ 4 & 5 & 6 & 13 & 14 & 15 \\ 7 & 8 & 9 & 16 & 17 & 18 \end{array}$$

The multi reduction in each dimension x is a vector of length 3 for $x = 0$ (horizontal) or $x = 1$ (vertical); for $x = 2$ (the outer dimension) the multi reduction is a vector of length 2. The multi reduction with operator + for each of the three dimensions is

$$\begin{array}{cc} x & result \\ 0 & [51 \quad 57 \quad 63] \\ 1 & [39 \quad 57 \quad 75] \\ 2 & [45 \quad 126] \end{array}$$

The leftmost element of the result for dimension $x = 1$ is given by $1 + 2 + 3 + 10 + 11 + 12$, the sum of all the elements in the two upper rows of Q. The center element of the result for dimension $x = 0$ is $2 + 5 + 8 + 11 + 14 + 17$, the sum of the two center columns of Q. The rightmost element of the result for dimension $x = 2$ is the sum of all the elements in the right half of Q.
The multi reduction with operator max in each of the three dimensions produce the vectors

$$\begin{array}{cc} x & result \\ 0 & [16 \quad 17 \quad 18] \\ 1 & [12 \quad 15 \quad 18] \\ 2 & [9 \quad 18] \end{array}$$

Figure 3.7: Multi Reductions on a $3 \times 3 \times 2$ Array

```
forall (x ∈ {1...k})
1. Load Computation Step in Dimension x
   forall (g ∈ {0...η_x(i)})                          loop over data indices
     W_x(i,g) ← max_x(A_{I,x,g})                      activity on data hyperplane (i,g)
   endfor
   S_i ← Σ_g(W_x(i,g))                                activity on hyperplane p_x(i)
   L_i ← Σ_{j<i}(S_j)                                 activity over hyperplanes p_x(j), j < i
   Ā ← (L_{p-1} + S_{p-1})/p                          broadcast to all processors
   if (Ā < η_x) then abort endif                      avoid underflows
   F_i ← iĀ − L_i                                     flow from hyperplane p_x(i) to p_x(i − 1)
2. Data Movement along Dimension x
   forall (i) in parallel
     while (F_i > W_x(i,0)/2)                         tested by each processor
       F_i ← F_i − W_x(i,0)
       put ()                                         put hyperplane D_x(i,0) to processors p_x(i − 1)
     endwhile
     W_p ← W_{i-1,η_x(i-1)}                           load of topmost data of p_x(i − 1)
     while (F_i ≤ −W_p/2)                             tested by each processor
       get ()                                         get hyperplane D_x(i − 1, W_p) into D_x(i,0)
       F_i ← F_i + W_x(i,0)
       W_p ← W_{i-1,η_x(i-1)}                         p_x(i − 1) may or may not call get ()
     endwhile
   endfor
endfor
```

Figure 3.8: Scan Directed Load Balancing for More than One Dimension

3.3.4 Data Movement Step

There are two major differences between Data Movement Steps for one and for several dimensions. The first is that `put()` and `get()` now shift hyperplanes of data rather than individual data points. The second is that data hyperplane loads $W_x(i, g)$ are no longer restricted to having a value of either 0 or 1.

The new definitions for `put()` and `get()` are careful to maintain a multi dimensional queuing discipline, so the order of data from different hyperplanes is maintained and the domain decomposition μ remains orthogonal and monotonic.

Definition 18 *The function* `put()` *places the hyperplane of data* $D_x(i, 0)$ *on every active processor hyperplane onto the processor hyperplane* P_{i-1} *at location* $\eta_x(i-1)$*. The values of* $\eta_x(i)$ *and* $\eta_x(i-1)$ *are updated to reflect the changes, and the data in processor hyperplane* P_{i-1} *has its index* g *reduced by one.*

Definition 19 *The function* `get()` *takes the hyperplane of data at* $D_x(i-1, \eta_x(i-1)-1)$ *for every active processor hyperplane* P_i *and places it into data* x*-hyperplane* 0*, which is freed by incrementing the index* g *of the existing data. The values of* $\eta_x(i)$ *and* $\eta_x(i-1)$ *are updated to reflect the changes.*

The new termination condition for the inner loops of the Data Movement Step reflects the fact that hyperplane loads may differ by substantial amounts. A hyperplane is shifted if the flow is at least half of the hyperplane's load. This condition is analogous to the one in the one dimensional algorithm and avoids shifting a hyperplane unless the shift will improve the load. This is analyzed and justified in Section 3.5.

3.4 Improvements

This section addresses several practical issues that are not addressed by the previous sections. The first problem is that the amount of storage per processor is limited, and could be exceeded by a distribution of data that places a very large number of inactive cells on a single processor. Since *sparse* layouts do not store inactive cells, this is only a problem with *dense* data layouts.

The possibility of decomposition underflow, in which one data point or data hyperplane is assigned to more than one processor or more than one processor hyperplane, threatens to violate the surjectivity property preserved by Scan Directed Load Balancing. A strategy for preserving surjectivity is presented and discussed.

Uneven amounts of data per processor can not only cause buffer overflow, but may also lead to uneven computational loads during the storage management phase. Therefore appropriate data structures for storing and accessing data are discussed next.

Finally, the advantages of using integer arithmetic for Scan Directed Load Balancing are presented with an example of the pitfalls of using floating point.

3.4.1 Buffer Overflows

The distinction between *dense* and *sparse* data layouts in a grid was discussed in Section 2.3: a dense problem is one in which inactive points in a grid take up storage in the processors. A typical example of this is the diffusion problem presented in Section 2.3, where each point may be inactive for some time, then be reactivated by a neighbor's changing value. When the pixel at coordinates H is reactivated, it should be in processor $P_{\mu(H)}$, so that processing may continue as if the point had always been active. In contrast, a sparse data set is one in which data only needs to be stored for active points; in a molecular simulation, for example, only those points of space that are active, i.e. that contain atoms, take up storage space, and all other points can safely be ignored.

For dense data sets, the load balancing described by the above algorithms sets no limit on the number of inactive data points that may end up in a single processor. If, for example, the data is evenly distributed in the machine, and all the data in the left half of the machine is active and all the data in the right half of the machine is inactive, exactly balancing the load distribution moves all the inactive data into the rightmost processor so the active points can be evenly distributed. This is shown in Figure 3.9. Overflows can be a real problem for dense data sets, since the per-processor memory of highly parallel computers is often limited.

What follows assumes, for simplicity, that each processor can hold at most b^k points and that any attempt to store more than b x-hyperplanes (for any axis x) in one processor may cause a buffer overflow and loss of data.

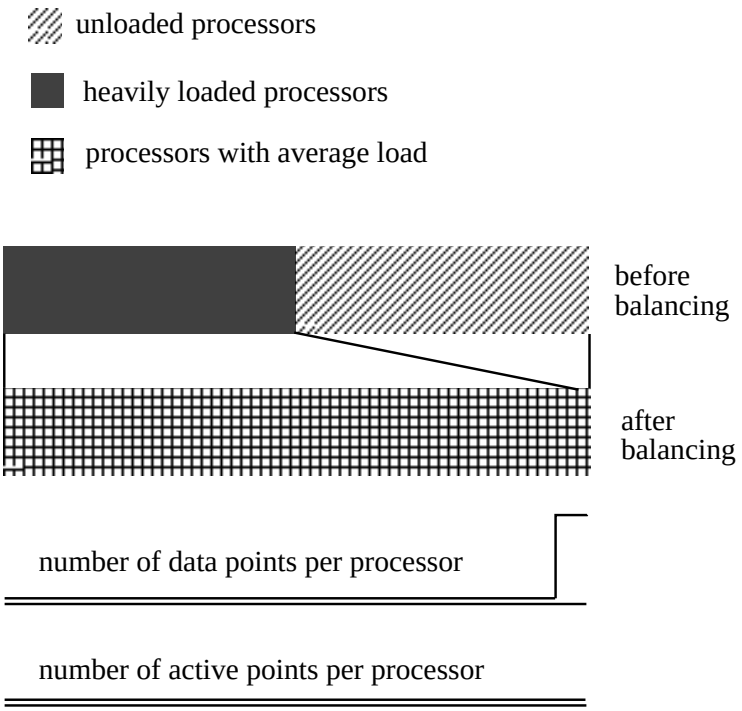


Figure 3.9: Perfect load balance requires large memories.

It is possible to use b^k storage locations to store more than b hyperplanes in one axis by storing fewer than b hyperplanes in some other axis, but for simplicity this possibility will not be analyzed further.

The next section presents two algorithms for dense data sets that avoid buffer overflows and provide good load balancing, though not necessarily as good as the algorithms of the previous section.

3.4.2 Avoiding Buffer Overflows

```
1. Load Computation Step
   L_i ← Σ_{j<i}(A_j)                          activity to left of processor i
   Ā ← (L_{p-1} + A_{p-1})/p                   Ā computed on processor p − 1
   if (Ā < 1) then abort load balancing endif   avoid underflows
   α ← Ā/(b − n/p)                             processor p − 1 computes, broadcasts α
   A'_i ← A_i + αD_i                           compute modified load
   L'_i ← Σ_{j<i}(A'_j)                        new load to left of processor i
   Ā' ← L'_{p-1}/p                             Ā' = Āb/(b − n/p)
   F_i ← iĀ' − L'_i                            flow from processor i to i − 1
2. Data Movement Step
   forall (i) in parallel
     while (F_i > (1 + α)/2)                   tested by each processor
       if active(D_{i,0}) then F_i ← F_i − (1 + α)
       else F_i ← F_i − α endif
       put ()
     endwhile
     while (F_i ≤ −(1 + α)/2)
       get ()
       if active(D_{i,0}) then F_i ← F_i + (1 + α)
       else F_i ← F_i + α endif
     endwhile
   endfor
```

Figure 3.10: One-Dimensional Load Balancing with Limited Buffers

The plain one dimensional algorithm expects to have unlimited buffering space for inactive data points; the one dimensional algorithm for bounded

buffers is similar, but insures that no processor is assigned more than b data points. This is achieved by balancing a *pseudo* load A' instead of the *actual* load A; here and in what follows, A' is the sum of the pseudo load on each processor, $A' = \sum_i A'_i$, and A is the total actual load, $A = \sum_i A_i$.

The pseudo load A' is a linear combination of the actual load A and the number n of data points:

$$A' = A + \alpha n = A(1 + \alpha) + (n - A)\alpha \tag{3.1}$$

The pseudo load of a data point is always greater, by a value α, than the point's actual load: as can be seen in Equation 3.1, active data points have a pseudo load of $\alpha + 1$ and inactive points (of which there are $n - A$) a pseudo load of α. The value of α can be selected to insure that no processor ever receives more than b data points, as shown in Figure 3.11.

If the load balancing were able to distribute load exactly, each processor would have a pseudo load of $\bar{A}'$ and the load would be guaranteed not to overflow. However, pseudo load is associated with data points, each of which carries a pseudo load of either α or $1 + \alpha$. Scan Directed Load Balancing places each data point in the processor that contains most of the data point's load or, if a data point is evenly split between two processors, it is always placed in the one with the lower index [1].

The following theorem shows that the value of α from Equation 3.2 limits the number of data points in each processor to no more than b even when, because load is associated with indivisible data points, load cannot be distributed so that each processor has exactly a load of $\bar{A}'$. In distributing the load, each data point is assigned to that processor which has most of its load; if two processors are assigned equal shares of a data point's load, the data point is always assigned to the processor with the lesser index. The load of a point can never be distributed among more than two processors, since the algorithm only re-assigns load to processors if $\bar{A}' \geq 1 + \alpha$.

Theorem 2 *If the pseudo load A' is equalized, so each processor is assigned a load of $\bar{A}' = A'/p$ and each data point has a pseudo load at least α, no processor will be assigned more than b points.*

[1] If the load of a point were distributed over more than two processors, the distribution would underflow; this situation is discussed in Section 3.4.3, and avoided by Scan Directed Load Balancing.

The value α is the (pseudo) load equivalent of a unit of storage, so the value A/α is the storage equivalent of the total load $A = \sum_i A_i$. Each of n points in the machine requires one unit of storage, and the total storage in the machine is fixed at bp; the condition

$$bp \geq A/\alpha + n$$

insures that the storage is sufficient for the n points plus the storage equivalent of the load A. The equation can be rewritten to yield a definition of α:

$$\alpha \geq \frac{A}{bp - n}$$

The smaller the α, the more closely the pseudo load A' approximates the actual load A; therefore, taking

$$\alpha = \frac{A}{bp - n} \tag{3.2}$$

gives the best approximation to the actual load while guaranteeing that storage will not overflow.

Figure 3.11: Derivation of α in one Dimension

Proof If a processor is assigned data points whose load adds up to exactly $\bar{A}'$, the theorem is proved in the derivation for α (Figure 3.11).

If some of the data points for which the processor is assigned load also have some load on a different processor, the processor may *share* data points with its upper neighbor, with its lower neighbor, or with both. An example of the latter case is shown in Figure 3.12.

In all these three cases, if the data point shared by processor P_i with the neighbor is actually assigned to the neighbor, the number of points on P_i is less than it would be if the data point were assigned to P_i; proving that the latter case does not overflow therefore also proves that the former does not overflow.

If processor P_i only shares with the upper neighbor P_{i+1} and the shared data point is assigned to P_i, the pseudo load of all the points that are entirely within P_i is at most $A'_i \leq \bar{A}' - \alpha/2 = \alpha(b - 1/2)$, since otherwise the shared point would be assigned to P_{i+1}. Since each of the points entirely on P_i contributes a pseudo load of at least α, the number of such points is at most $\lfloor b - 1/2 \rfloor = b - 1$, and the total number of points is b or fewer.

If processor P_i only shares with the lower neighbor P_{i-1} and the shared data point is assigned to P_i, the pseudo load of all the points that are entirely within P_i is at most $A'_i < \bar{A}' - \alpha/2 = \alpha(b - 1/2)$, since otherwise the shared point would be assigned to P_{i-1}. Since each of the points entirely on P_i contributes a pseudo load of at least α, the number of such points is at most $b - 1$, and the total number of points is b or fewer.

If processor P_i shares a data point with both its upper and its lower neighbor and both the upper and lower data points are assigned to P_i, the portion of the pseudo load of P_i that is contributed by the lower shared data point is $A'_l \geq \alpha/2$, and that contributed by the upper shared data point is $A'_u > \alpha/2$; the sum is $A'_u + A'_l = A'_{ul} > \alpha$. The pseudo load of all the points that are entirely within P_i is therefore at most $A'_i < \bar{A}' - \alpha = \alpha(b - 1)$. Because $A'_i < \alpha(b - 1)$ and the minimum load of a data point is α, there are at most $b - 2$ data points entirely on P_i, and the two shared data points do not cause overflow. □

The main difference between the Data Movement phases of the bounded and of the unbounded buffer algorithm is the amount by which the flows, F_i, are updated when shifting out an active or an inactive data point. The terminating condition is also updated so that a load is only transferred if the

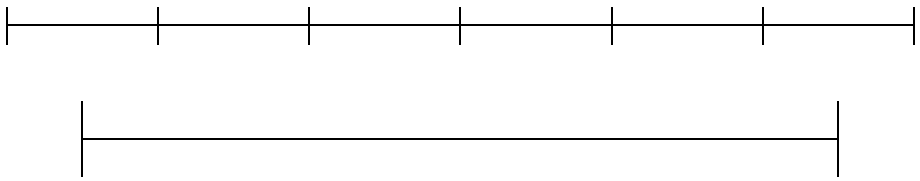


Figure 3.12: Non-integral Number of Points Assigned to a Processor

flow requires at least one more active point.

The multi dimensional bounded buffer algorithm is simply the multi dimensional extension of the one dimensional bounded buffer algorithm, and quite similar to the multi dimensional unbounded buffer algorithm. Each processor is assumed to have a k dimensional buffer for data of size b^k. The computed hyperplane weights $W'_{i,g}$ are used during shifting instead of the simpler α or $1+\alpha$ data point weight used in the one dimensional bounded buffer case, and the terminating condition for the shift is modified accordingly.

3.4.3 Buffer Underflows

A practical algorithm for load balancing must insure not only that buffer overflow does not occur, but also that buffer underflow is handled properly. Underflow would violate the Surjectivity of an OMS distribution, and can happen whenever the average load per processor is less than one active data point or, for $k > 1$, less than ℓ, the maximum load of a data hyperplane. To visualize underflow, imagine a computation, on a processor array of dimension one, which only has two remaining active points, and these points are neighbors. Scan Directed Load Balancing attempts to assign the left point to all of the left half of the machine, and the right point to all of the right half of the machine. The two processors that the two points end up being stored in are unlikely to be neighbors.

It is to avoid underflows that Scan Directed Load Balancing refuses to

```
forall (x ∈ {1...k})
1. Load Computation Step in Dimension x
   forall (g ∈ {0...η_x(i)})                          loop over data indices
     W_x(i,g) ← max_x(A_{I,x,g})                      activity on data hyperplane (i,g)
   endfor
   S_i ← Σ_g(W_x(i,g))                                activity on hyperplane p_x(i)
   L_i ← Σ_{j<i}(S_j)                                 activity over hyperplanes p_x(j), j < i
   Ā ← (L_{p-1} + S_{p-1})/p                          broadcast to all processors
   if (Ā < η_x) then abort endif                      avoid underflows
   α ← Ā/(b − n/p)                                    p_x(p) compute, broadcast α
   forall (g ∈ {0...η_x(i)})
     W'_{i,g} ← W_x(i,g) + α                          new load for D_x(i,g)
   endfor
   S'_i ← Σ_g(W'_{i,g})                               new load for p_x(i)
   L'_i ← Σ_{j<i}(S'_j)                               new load to left of p_x(i)
   Ā' ← (L'_{p-1} + S'_{p-1})/p                       Ā' = Āb/(b − n/p)
   F_i ← iĀ' − L'_i                                   flow from hyperplane p_x(i) to p_x(i − 1)
2. Data Movement along Dimension x
   while (F_i > W'_{i,0}/2)                           tested by each processor
     F_i ← F_i − W'_{i,0}
     put ()                                           put hyperplane D_x(i,0) to processors p_x(i − 1)
   endwhile
   W_p ← W'_{i-1,η_x(i-1)}                            load of topmost data of p_x(i − 1)
   while (F_i ≤ −W_p/2)                               tested by each processor
     get ()                                           get hyperplane D_x(i − 1, W_p) into D_x(i,0)
     F_i ← F_i + W_p                                  W_p is the new load shifted in
     W_p ← W'_{i-1,η_x(i-1)}                          p_x(i − 1) may or may not call get ()
   endwhile
endfor
```

Figure 3.13: Multi Dimensional Load Balancing with Limited Buffers

change the decomposition μ if the average load is less than ℓ. Because underflow can only occur when the average load is low, the maximum load is likely to be low also, and balancing the load probably would not substantially affect the total time of the computation anyway, so not balancing the load is an appropriate action to take. The detection of potential underflows is fast and inexpensive, and is not likely to substantially slow down most computations.

Because of the lack of practical motivations for doing so, alternative approaches have not been explored in depth in this thesis. Possible approaches include using pseudo load for inactive points to raise the average pseudo load above ℓ, and allowing the decomposition to not be surjective. If only active points communicate, the latter approach may be profitable because, when the average load is low, the total communication bandwidth required is also small, and local communication is not as crucial to performance as when the average load is high.

3.4.4 Efficient Management of Active Processes

A complete algorithm for load balancing should include an efficient mechanism for keeping track of the active data points on each processor. This is especially important for *dense* layouts, in which the number of points stored by each processor is potentially much greater than the number of active points, but is also relevant for *sparse* layouts, which need a good data structure to avoid wasting space on grid points that are not associated with any data point.

The data structure that organizes the data points contained in each processor must support five operations on the data structure. Two are the `put()` and `get()` functions, which add and remove hyperplanes of data points from any one of the $2k$ ends of the data structure. Two more operations are to *add* or *remove* a data point from the structure; this is needed when a point becomes active or inactive. The *add* operation only needs to be able to efficiently add a neighbor of an active point, since in a local computation data points that are not neighbors do not communicate. The fifth operation is simply the *traversal* of the active list, in some order that may be defined by the algorithm. In what follows, $A_{max} = \max_I(A_I)$ is the largest load on any processor.

For a one dimensional array of data and processors, this data structure can be a simple doubly-linked list that keeps the active data points sorted in

This code runs in parallel on all processors. The variable `Sending` is true on those processors for which $F_i > threshold$, the variable `Receiving` is true on those processors for which $F_{i+1} > threshold$. `Leftactive` and `Rightactive` are the head and tail of the doubly-linked list on each processor.

```
if Sending then
 if Leftactive = Points [Left] then
  Leftactive := Leftactive->Right;
  if Leftactive # NIL then
   Leftactive->Left := NIL;
  endif
 endif
 send-to-left (Points [Left], NewPoint);
 send-to-left (Activity [Left], NewAct);
 Left := Left + 1;
endif
if Receiving then
 Right := Right + 1;
 Points [Right] := NewPoint;
 Activity [Right] := NewAct;
 if NewAct > 0 then
  Points [Right]->Right := NIL;
  Points [Right]->Left := Rightactive;
  Rightactive->Right := Points [Right];
  Rightactive := Points [Right];
 endif
endif
```

Figure 3.14: One dimensional `put()` using doubly-linked lists

order of increasing value of the coordinate. All active points are kept in the doubly-linked list; when an active point is added to or removed from either end, the corresponding pointers are updated. This approach requires a small, constant amount of time and space per data point and allows easy traversal of the list during processing. Putting and getting are $O(1)$, and so is activating a neighbor of an active point or removing an active point. Traversal can be done in $O(A_{max})$ in list order. The code for `put()` using this data structure is shown in Figure 3.14; the code for `get()` is analogous. Different approaches using trees, especially balanced trees such as AVL trees [1] or B-trees [23] require more work and are therefore slower without providing any asymptotic improvement for these five operations.

For k dimensional load balancing, the `put()` and `get()` operations need to access entire hyperplanes at either end of each axis x, which suggests a geometrical approach such as that of quadtrees and octrees [82] or even the Grid File system [42] [44]. These approaches, however, are quite complicated and therefore require either substantial space or time for execution; furthermore, they are designed to store and encode information for a fixed data or coordinate space. Each processor in Scan Directed Load Balancing stores a varying portion of the data space, and a geometric trees or a Grid File would end up building, in each processor, a data structure for the data of the entire machine; this structure may take up precious storage space, and its traversal may not end up being much faster than a traversal of the entire data space.

A different approach again uses doubly-linked lists, which are well-suited also to keeping track of the activity of data points in multi-dimensional Scan Directed Load Balancing. Each active point belongs to k doubly-linked lists, where k is the grid and mesh dimension. Each active point has two pointers in each dimension, one to the next active point in each direction. Those active points for which there is no active point in a given direction have a sentinel value for the corresponding pointer. As a result, there are $(\Pi_{y \neq x} \eta_y)$ doubly-linked lists for *each* dimension x, one list for each possible point position on a hyperplane that is perpendicular to axis x.

When a hyperplane of data is shifted across a processor boundary in a `put()` or `get()` operation on dimension x, the $2k-2$ links that each active point has in dimensions orthogonal to x are still valid, and for each active point only the 2 links along the axis of the shift must be updated. For a fixed dimension k, each active point has a constant number of pointers, so the time for updating these links is proportional to the number of active

points, which can be located using the links that lie within the hyperplane. The time for removing an active point is $O(k)$; the time for adding an active point can be greater, since each point needs to be added to k independent lists. However, with a hyperspace-sweep mechanism that is a generalization of standard plane-sweep and space-sweep [43], it should be possible to sweep through all the active points in time $O(kA_{max} \log A_{max})$ and insert any newly active points into the data structure.

If a processor can store up to b^k points, the storage required for the linked lists is $2kb^{k-1}$ pointers for the heads and tails of the doubly-linked lists and $2kb^k$ pointers within the data structure. For fixed k, this is linear in b^k. Since in the worst case any optimal data structure will have to store up to b^k points, this algorithm is at most a constant factor more expensive than an optimal data structure.

3.4.5 Computing with Integers

A final consideration in a practical load balancing algorithm is that of floating point imprecision. The inputs to the algorithm are the loads of each data point or data hyperplane, and these are exact integers; the two one-dimensional algorithms have one division each, the multi dimensional algorithms each have three divisions. If these divisions are done in floating point arithmetic, the following problem can arise due to floating point inaccuracy. Consider for example the one dimensional load balancing illustrated by Figure 3.15, in which processor P_1 has $b = 24$ inactive data points and is therefore unable to accept any more data.

In this example the average load per processor is $\bar{A} = 8$. The number of data points per processors is $n/p = 12$ and the available buffer space is up to $b = 24$ data points. The value of $\alpha = \bar{A}/(b^k - n/p) = \frac{2}{3}$ means that the average pseudo load per processor is $\bar{A}' = 8 + 12 \times \frac{2}{3} = 16$.

Processor P_0, at the far left of Figure 3.15, has 1 inactive data point (shown as a square) and 9 active points (shown as circles), which gives $A'_0 = 9 + 10 \times \frac{2}{3} = 15\frac{2}{3}$ and $F_1 = 16 - 15\frac{2}{3} = \frac{1}{3}$.

Processor P_1 has $b = 24$ inactive points, for a total pseudo load of $A'_1 = 24 \times \frac{2}{3} = 16$. This means that $F_2 = F_1 = \frac{1}{3}$, and processor P_1 expects to receive $\frac{1}{2}$ inactive data point from P_2 and to give $\frac{1}{2}$ inactive data point to P_0. Any consistent rounding strategy will make sure that P_1 gives and receives either one or no data points, since receiving one data point from P_2 without

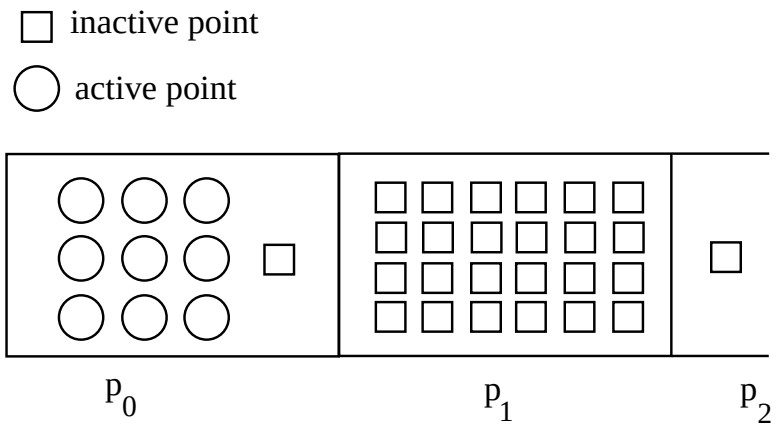


Figure 3.15: Example of Floating Point Arithmetic Causing Buffer Overflows

giving one to P_0 causes overflow in P_1, which has no free space.

When using floating point, however, the inaccuracy of floating point computation may cause F_1 and F_2 to be $F_1 = 16 - 9 + 10 \times \frac{2}{3} = \frac{\alpha}{2} - \epsilon$ and $F_2 = 32 - 15\frac{2}{3} - 24 \times \frac{2}{3} = \frac{\alpha}{2} + \epsilon$, even though mathematically $F_1 = F_2$. One data point will therefore be shifted from P_2 to P_1 and none will be shifted from P_1 to P_0, which causes overflow in P_1. One easy solution to this problem is to use a buffer size of $b - 1$ when computing the distribution, since the overflow can be at most 1. A cleaner solution is to use integer or rational arithmetic instead of floating point.

To use integers instead of floating point numbers, each division in each of the algorithms for Scan Directed Load Balancing is replaced by a rescaling of all the relevant quantities. The multi dimensional algorithm has three divisions, by p, by $b - n/p$, and again by p, so the variables used in the computation must be able to represent at least $\bar{A}'p^2(b-n/p)$. For the diffusion problem described in Section 2.3 on the 64×64 processor Maspar MP-1, $p = 64$, $b = 16$, $n/p \geq 4$, and $\bar{A}' \leq 72$, with a maximum value of $3,538,944$ which is easily represented using 32-bit integers.

The next section analyzes the theoretical performance of Scan Directed Load Balancing, and the following two chapters take a look at the performance of Scan Directed Load Balancing on real problems.

3.5 Performance

This section shows that the time required for executing the load computation step, the component of any load balancing algorithm most commonly analyzed in the literature, is $O(k(b^k + b \log p))$ if scans are implemented in logarithmic time, and $O(k(b^k + bp))$ otherwise. This time is expressed in terms of k, the dimension of the mesh, b, the maximum buffer size in each dimension, and p, the number of processors in each dimension.

3.5.1 Load Balance Achieved

The algorithm for one dimensional load balancing with unbounded buffers given in Figure 3.2 computes values for F_i, the optimal flow of active points from right to left (from left to right if $F_i < 0$) across the boundary between processors P_i and P_{i-1}. Since $F_i = i\bar{A} - L_i$ and $L_i = \sum_{j<i} A_i$, $F_{i+1} = F_i - A_i + \bar{A}$ also holds. Processor P_i sends out f_i active data points and receives f_{i+1} data points, with each $f_i = \lceil F_i - 0.5 \rceil$[2]. After the shift, processor P_i will have $A_i^{t+1} = A_i + f_{i+1} - f_i$ active points, with A_i^{t+1} the value of A_i on the next iteration.

The definitions of f_i and f_{i+1} give $A_i^{t+1} = A_i + \lceil F_{i+1} - 0.5 \rceil - \lceil F_i - 0.5 \rceil$, which becomes $A_i^{t+1} = A_i + \lceil F_i - A_i + \bar{A} - 0.5 \rceil - \lceil F_i - 0.5 \rceil$. Since A_i is an integer, this reduces to $A_i^{t+1} = \lceil F_i + \bar{A} - 0.5 \rceil - \lceil F_i - 0.5 \rceil$. If $\bar{A}$ is an integer, $A_i^{t+1} = \bar{A}$. If $\bar{A}$ is not an integer, A_i^{t+1} is still the difference of two integers and therefore an integer, and can be at most $A_i^{t+1} < \bar{A} + 1$, since $\lceil a + x \rceil - \lceil x \rceil < a + 1$ (and taking $a = \bar{A}$ and $x = F_i - 0.5$); therefore, $A_i^{t+1} < \bar{A} + 1$ and so $A_i^{t+1} \leq \lceil \bar{A} \rceil$. Reasoning along similar lines gives that $A_i^{t+1} \geq \lceil \bar{A} \rceil - 1 = \lfloor \bar{A} \rfloor$. Since A_i^{t+1} is an integer, either $A_i^{t+1} = \lceil \bar{A} \rceil$, or $A_i^{t+1} = \lfloor \bar{A} \rfloor$.

The number of active points assigned to any processor by this algorithm therefore obeys Equation 3.3:

$$\lfloor \bar{A} \rfloor \leq A_i^{t+1} \leq \lceil \bar{A} \rceil \tag{3.3}$$

This distribution of activity is optimal, since no processor has more than $\lceil \bar{A} \rceil$ active points.

[2]This simple formula for f_i is achieved by making the loop termination conditions asymmetric for $F_i > 0$ and $F_i < 0$ in the data movement step. Symmetrical conditions would give a formula for f_i that is dependent on whether $F_i \geq 0$.

The multi dimensional unbounded buffer algorithm is similar to the corresponding one dimensional algorithm in that the total activity of the hyperplanes sent out along axis x is f_i. However, the load of a hyperplane can be anywhere between 0 and $\ell = \max_{i,g}(W_x(i,g))$, whereas for the previous algorithm it was constrained to be either 0 or $\ell = 1$. A worst-case bound on the distribution of active points can be computed by assuming that, whenever the data movement loop ends, $W_{i,0} = \ell$ and $W_p = \ell$, giving $F_i - \ell/2 \leq f_i \leq F_i + \ell/2$. Since the same formulas as above hold for F_{i+1} and A_i^{t+1}, this algorithm obeys Equation 3.4:

$$\bar{A} - \ell \leq A_i^{t+1} \leq \bar{A} + \ell \tag{3.4}$$

This load balance may not be optimal, since the load on any single processor may be zero, and the load on the most heavily loaded processor may be as much as $\bar{A} + \ell$ and $\bar{A} = A/p$. This possibly imperfect load balance is one consequence of the orthogonality of the decomposition.

3.5.2 Load Balance with Limited Buffers

Unlike the one dimensional unbounded buffer algorithm, the bounded buffer algorithm only guarantees an optimal distribution of pseudo load, so that

$$\bar{A}' - \ell \leq A_i'^{t+1} \leq \bar{A}' + \ell \tag{3.5}$$

where for this algorithm $\ell = 1 + \alpha$. This equation is identical to Equation 3.4 (except for replacing A by A') and applies whenever the data points or hyperplanes do not all have the same load; in this case, the data points have load either α or $1 + \alpha$. The definition of $\bar{A}'$, $\bar{A}' = \bar{A} + \alpha n/p$, follows from the definition of A' in Equation 3.1. If each active point has a load of 1, the pseudo load of an active point is $1 + \alpha$ and (referring back to Figure 3.11) $A_i \leq A_i'/(1 + \alpha)$. Since $\alpha = \bar{A}/(b - n/p)$, large buffers $b \gg n/p$ make α small and the performance of this algorithm is comparable to that of algorithm with unlimited buffers; if only small buffers are available, n/p is close to b and the maximum load on any processor may be much higher than the average load.

The exact bound for A_i^{t+1} in terms of $\bar{A}$, α, and n/p is given by equation 3.6,

$$0 \leq A_i^{t+1} \leq \frac{\bar{A} + 1 + \alpha(n/p + 1)}{1 + \alpha} \tag{3.6}$$

which can be reformulated using the definition (Equation 3.2) of α to exclude the factor α and give

$$0 \leq A_i^{t+1} \leq 1 + \bar{A}\frac{b}{\bar{A} + b - n/p} \tag{3.7}$$

A_i^{t+1} is 0 when P_i has b inactive points, and increases to at most $\bar{A}\frac{b}{\bar{A}+b-n/p}$ when it has all active points.

If most data points are active, $\bar{A} \approx n/p$ and the bound reduces to $A_i^{t+1} \leq \bar{A} + 1$, which is close to the bound for the algorithm with unbounded buffers. The worst-case performance is obtained when $b \approx n/p$, in which case $A_i^{t+1} \leq b + 1$, the best bound that can be given, simply means little redistribution is taking place and a processor may happen to have all its n/p data points active. This corresponds to intuition, since if little or no buffer space is available, load balancing is unable to contribute much to performance.

Analysis of the multi dimensional bounded buffer algorithm produces similar results, the main difference between the two being that in the multi dimensional algorithm the load of an active hyperplane can range anywhere between $1+\alpha$ and $\ell+\alpha$. The pseudo load is bounded by $A_i'^{t+1} \leq \bar{A}' + (\ell + \alpha)$, so the load is bounded by $A_i \leq A'_i\ell/(\ell + \alpha)$. The same reasoning as above gives

$$A_i^{t+1} \leq \frac{\bar{A} + \ell + \alpha(n/p + 1)}{1 + \alpha} \tag{3.8}$$

$$A_i^{t+1} \leq \frac{\bar{A}(b + 1) + \ell(b - n/p)}{\bar{A} + b - n/p} \tag{3.9}$$

3.5.3 Satisfying the Locality Property

The next theorem shows that the load balancing algorithms presented above preserve Orthogonal, Monotonic, and Surjective (OMS) data decompositions.

Lemma 1 *The* `put()` *and* `get()` *operations preserve the orthogonality of* μ.

Proof This is trivially true in one dimension. If $k > 1$, consider the case in which some processor P_i is doing a `put()` operation in dimension x. Then, since F_i and $W_{i,0}$ are identical by construction for all processors P_i, all the

data points at coordinate $(i, 0)$ are transferred together and the μ remains orthogonal. Correspondingly for `get()`. □

Lemma 2 *The* `put()` *and* `get()` *operations preserve the monotonicity of* μ.

Proof Definition 18 states that `put()` transfers the leftmost data hyperplane from one processor onto the rightmost end of the data buffer of the neighbor to its left; this preserves the orthogonality of μ, since each buffer in each processor acts like a FIFO queue and no data point can ever "pass" another data point. The same is true of `get()`, which proves the lemma. □

Lemma 3 *Scan Directed Load Balancing preserves the surjectivity of* μ.

The following proof only explicitly presents the one dimensional case with unlimited buffering. To extend the proof to more than one dimension, replace the term "data point" with "data hyperplane", and the term "processor" with "processor hyperplane"; to extend to the limited buffer algorithms, replace "load" with "pseudo load" and "$\bar{A}$" with "$\bar{A}'$".

Proof In Scan Directed Load Balancing, balancing is only done if the average load $\bar{A}$ assigned to a processor is less than the maximum load ℓ of a data point, so if $\bar{A} > \ell$, μ is unchanged and surjectivity is preserved.

If $\bar{A} \geq \ell$, μ will be changed so each processor is assigned a load of approximately $\bar{A}$. More accurately, the load of each data point is assigned to one or two processors, and the data point is actually assigned to the processor to which the larger part of its load is assigned, or to the processor with the lower index if the load of the point is equally divided among two processors. If a data point D_h with load ℓ is equally divided between processors P_{i-1} and P_i, it is actually assigned to P_{i-1}. The portion of the load of P_i that is not from D_h is $\bar{A} - \ell/2 \geq \ell/2$, which means P_i will always be assigned at least the next point D_{h+1}. As a result, each processor has at least one data point and surjectivity is preserved. □

Lemma 4 *Scan Directed Load Balancing preserves the Orthogonal, Monotonic, and Surjective decompositions.*

Proof Assume that the domain decomposition to which the algorithm is applied is already Orthogonal, Monotonic, and Surjective. Since Scan Directed Load Balancing uses `put()` and `get()` to perform all data movement, by Lemmas 1 and 2 Scan Directed Load Balancing preserves Orthogonality and Monotonicity. By Lemma 3, Scan Directed Load Balancing also preserves Surjectivity, so the Orthogonal, Monotonic, and Surjective properties of the domain decomposition are preserved by Scan Directed Load Balancing. □

3.5.4 Execution Time of Load Computation Step

Lemma 5 *A scan over a one dimensional computer of size p can be done in time $O(\log p)$ using a binary tree of processors.*

This has been shown by Ladner and Fischer [57, 58].

The binary tree of processors need not be a separate computing element. In hypercube-connected parallel computers, a binary tree can be embedded directly into the hypercube, with the parallel computer's processing elements performing the combining operation. On level i of the tree, the children of a node are the same node, and the node that differs from it by bit i of the address. A mesh-connected one dimensional computer could use p additional long connections to organize its processors into a tree. The total area of these connections would be $O(p)$, and the cost of the tree organization is therefore no greater than that of the mesh.

It is also possible to use the mesh to execute the scans in $\log p$ steps, pipelining the shifts to obtain good performance. Usually when doing communication on a mesh, the time required for sending the data along the wires is only a fraction of the total time required for communication, since time must be spent buffering the data and, for irregular communications, routing, arbitrating, and queuing messages. When using the mesh to execute a scan, every communicating processor at every step of the scan is using dedicated wires that are not required by any other processor. This means the scan communication can be *pipelined*, which gives a great performance improvement over the normal shift speed. The Maspar MP-1 uses this approach to make scan operations faster than general shifts of distance p. This algorithm for scans does not scale as well as the approach using binary trees, since it is ultimately $O(p)$. This asymptotic behavior is unfortunate on one dimensional

linear arrays of processors, but scales well on multi dimensional meshes with p^k processors, in which the time for scans increases much more slowly than the number of processors.

Definition 20 *A* Scan Time S *is defined as the maximum time required to perform a plus-scan or a max-scan in either direction along one dimension of a mesh-connected computer. Depending on the available hardware,* S *can be* $O(\log p)$ *or as much as* $O(p)$*.*

Lemma 6 S *is* $\Omega(\log p)$*.*

Proof Each scan over p processors requires at least $\log(p-1)$ combining operations to produce the last value, which is the sum of the preceding $p-1$ values. Even if these operations are pipelined, the delay of the pipeline is still $\Omega(\log p)$. □

The time required to compute a scan affects the amount of time required to complete an iteration of Scan Directed Load Balancing. The primitive communication operations in Scan Directed Load Balancing are scans, broadcast operations, and multi reductions.

Lemma 7 *The time required for a broadcast operation is* $O(S)$*.*

Proof A broadcast operation from processor P_i to all other processors can be implemented by two plus-scans in opposite directions. P_i uses as the argument for both scans the value (v) to be broadcast, and every other processor uses the identity value 0. The result of the left-to-right scan is 0 for every $P_j, j \leq i$, and v for every $P_j, j > i$. The same is done with the right-to-left scan, and the result is v for every $P_j, j < i$, 0 for every $P_j, j \geq i$. Every processor finally adds its two scan results and its value, which gives v on every processor. □

A scan is an operation over a linear array of data, so the a scan over a multi dimensional data set must linearize its data, that is, impose a total ordering over all the elements of the data. For implementing the broadcast operation, the ordering is irrelevant as long as it is consistent, so that if in a right-to-left scan $D_H <_\sigma D_{H'}$ for some data elements D_H and $D_{H'}$,

then $D_{H'} <_\sigma D_H$ for the corresponding left-to-right scan, with $<_\sigma$ the total ordering.

For implementing functions other than the broadcast operation, the particular linearization used can be significant. This dissertation only uses *orthogonal* linearizations. An orthogonal linearization always imposes a total ordering on the dimensions $x_0 \ldots x_{k-1}$ so that if $i \neq j$, either x_i *precedes* x_j, or x_j precedes x_i. An orthogonal linearization is defined by $D_{H'} <_\sigma D_H$ holding if and only if both $f(H_{x_i}) < f(H'_{x_i})$ for some dimension x_i and some function f, and $H_{x_j} = H'_{x_j}$ in all dimensions x_j that precede x_i. In this dissertation, the function $f(H_x)$ is either H_x or $-H_x$. The dimension which precedes every other dimension is the *major* dimension for the scan.

For example, if $k = 2$ there are exactly 4 distinct orthogonal scans as described above. In two of them, distinguished by $f(H_x) = H_x$ and $f(H_x) = -H_x$, the rows are the major dimension, and the data is linearized by placing columns end-to-end to form a single very long column. For the other two scans the columns are the major dimension, and the data is linearized by placing rows next to one another.

Lemma 8 *The time required for a multi reduction with operator* max *is* $O(S)$.

Proof A multi reduction $\rho(x, \max, Q)$ can be implemented by at most three scans; the last scan has x as the major dimension. The first scan computes the maximum value of the input, $Q_m = \max(Q)$, and the second scan, in the reverse direction, broadcasts Q_m to all elements. Each element with index H then adds $H_x \times (Q_m + 1)$ to its value; since each element of $\rho(x, \max, Q)$ is at most Q_m, this effectively isolates the computation in each hyperplane from those in the other hyperplanes. Subtracting $H_x \times (Q_m + 1)$ from the result of the last scan gives the multi reduction $\rho(x, \max, Q)$. □

Simpler implementations with only two scans are available if the operator of the multi reduction is invertible.

The value η_x is the maximum number of data hyperplanes per processor in dimension x. The value $\eta = \max_x \eta_x$ is the maximum value of data hyperplanes per processor in any dimension.

Theorem 3 *A single Load Computation step for any one of the Scan Directed Load Balancing algorithms requires time* $O(S\eta + \eta^k)$.

Proof Each processor has at most η^k data points in it, and may require up to η^k constant-time operations such as addition or division in computing the per-processor hyperplane loads $A_{I,x,g}$. Computing the hyperplane loads $W_x(i,g)$ takes η multi reductions, computing L_i takes a single scan, and computing F_i from L_i takes a broadcast operation; computing a pseudo load at most doubles the number of scans required. The two terms $S\eta$ and η^k dominate the asymptotic time required by the Load Computation step, which is therefore $O(S\eta + \eta^k)$. □

Note that for the bounded buffer case, $\eta \leq b$, so $O(Sb + b^k)$ is also an accurate, if looser, bound.

Since there are k Load Computation steps when load balancing in k dimensions, the total time spent balancing the load is $O(k(S\eta + \eta^k))$.

3.5.5 Execution Time of Data Movement Step

Definition 21 *A* put time *PG is the maximum time required to execute a* `put()` *or a* `get()` *operation.*

Lemma 9 *The execution time PG of the* `put()` *and* `get()` *operations is bounded by* $O(\eta^{k-1})$.

Proof Definitions 18 and 19 state that the two functions performed by `put()` and `get()` are to transfer an x-hyperplane of $\Pi_{y \neq x}\eta_y$ data points from every active processor to its neighbor and to shift the remaining data in the processor by one position. Transferring one point is a constant-time operation, as analyzed in Section 3.4.4. If the data is stored in k dimensional circular buffers (this is possible if the stride is fixed for each dimension), shifting the data that remains in the processor can be achieved in time $O(\Pi_{y \neq x}\eta_y)$. Since $\Pi_{y \neq x}\eta_y \leq \eta^{k-1}$, the execution time of either `put()` or `get()` is $O(\eta^{k-1})$. □

The Data Movement step of the algorithms which balance load A is not as easily analyzed as that of the bounded buffer algorithms, which balance pseudo load A', because in the latter each data point or data hyperplane has a nonzero pseudo load $A' \geq \alpha$. Since the two loops in the Data Movement step shift points or hyperplanes until F_i is zero, and F_i is decremented by the (pseudo) load of the points shifted, if points may have zero load there is no bound (except for n) on the number of iterations of the loop. The following

theorem therefore only addresses the bounded buffer form of Scan Directed Load Balancing.

Theorem 4 *Each of the Data Movement steps in the bounded buffer algorithms is* $O(\eta^{k-1}\max(|F_i|))$.

Proof In the first half of the Data Movement step, all $F_i > W'_{i,0}$ are reduced by $W'_{i,0}$ on every iteration of the loop. Since $W'_{i,g}$ is at least $\alpha = \bar{A}/(b - n/p)$, it takes at most F_i/α steps, each requiring time PG, to transfer a pseudo load equal to $F_i - W'_{i,0}/2$. The same reasoning applied to those processors for which initially $F_i < 0$ gives that the number of loops in each Data Movement step is $O(\eta^{k-1}\max(|F_i|))$. □

Since in these algorithms $\eta \leq b$, each step can also be bounded by $O(b^{k-1}\max(|F_i|))$.

Since load balancing in k dimensions requires k Data Movement steps, the total time required for data movement in the bounded buffer algorithms is $O(kb^{k-1}\max(|F_i|))$.

Theorem 5 *Scan Directed Load Balancing with limited buffers executes in time no greater than* $O(k(S\eta + \eta^k + \eta^{k-1}\max(|F_i|)))$.

The proof follows immediately from the time bounds for the Load Computation and the Data Movement steps.

Since $b \geq \eta$ is finite and limited by the size of the per-processor memory, it can be taken to be a constant and yield a simpler bound of $O(k(S + \max(|F_i|)))$. The time S required for a scan directly affects the time required to balance the load, and so does the maximum flow across any one boundary. The time S can be assumed to be fixed for any given system, whereas the value of $\max(|F_i|)$ is heavily dependent on data distribution and on the frequency of load balancing: frequent load balancing usually means that, even before balancing, the load is nearly balanced, so $\max(|F_i|)$ is likely to be small; if load balancing is infrequent, $\max(|F_i|)$ may be large and load balancing may be slower than in the more frequent case.