Cost-Effective Data-Parallel Load Balancing

James P. Ahrens and Charles D. Hansen[1]

Department of Computer Science and Engineering
University of Washington

Technical Report 95–04–02
April 1995

[1] Advanced Computing Laboratory, Los Alamos National Laboratory, Los Alamos 87545

Cost-Effective Data-Parallel Load Balancing

James P. Ahrens
Dept. of Computer Science & Engineering
University of Washington
Seattle, Washington 98195

Charles D. Hansen
Advanced Computing Laboratory
Los Alamos National Laboratory
Los Alamos, New Mexico 87545

Abstract

Load balancing algorithms improve a program's performance on unbalanced datasets, but can degrade performance on balanced datasets, because unnecessary load redistributions occur. This paper presents a *cost-effective* data-parallel load balancing algorithm which performs load redistributions only when the possible savings outweigh the redistribution costs. Experiments with a data-parallel polygon renderer show a performance improvement of up to a factor of 33 on unbalanced datasets and a maximum performance loss of only 27 percent on balanced datasets when using this algorithm.

1 Introduction

Load balancing algorithms provide the basis for efficient parallel solutions to many important computational problems including the n-body problem, polygon and volume rendering, and optimization problems. The completion time of these parallel solutions depends on the completion time of the processor with the maximum computational workload. Load balancing algorithms attempt to distribute the computational workload evenly among all processors. This reduces the maximum workload on any processor and thus reduces the completion time of the parallel solutions.

In this paper, a data-parallel load balancing algorithm is described. The algorithm balances fine-grained computations within a data-parallel program. The workload balanced by the algorithm is not known until execution time, but once known does not change dynamically during the computation. The algorithm can be further characterized by when it balances; it balances only when it is cost-effective. A prediction of the load balancing costs and measurement of the possible savings are used to make this decision. A major advantage of utilizing a cost-effective load balancing algorithm is that the execution time of a load-balanced version of a program is never significantly worse than the execution time of the original version of the same program. This result depends only on never underestimating the costs of load balancing.

Wikstrom et al. [WPG91] use a computational model and experimental results to present evidence that using a load balancing algorithm does not always improve a program's performance. The authors show the execution time of a load-balanced version of a program can substantially exceed the execution time of the original version of the same program. This is because the costs of doing the balancing can exceed the savings achieved by the balancing. The authors identify concerns which trouble the users of most load balancing algorithms: Should this load balancing algorithm be used? Do the performance benefits of using the algorithm outweigh the performance degradations? An important contribution of this paper is the presentation of a load balancing algorithm which alleviates these concerns because the algorithm does not significantly degrade performance on any dataset.

Other researchers have studied the problem of deciding when to balance with different workloads and problem types. Nicol and Townsend [NT89] describe a performance model for parallel computations which assigns processors different partitions of an irregular grid. The model utilizes performance measurements of portions of the parallel computation and knowledge about how the grid structure will change over time to calculate an optimal schedule of grid repartitionings. One disadvantage of this scheme is that the computation of the schedule adds to execution time of the grid calculation. Nicol and Reynolds [NJ90] describe a load

balancing algorithm which is targeted for data-parallel applications with uncertain behavior, such as, the addition of work during execution and uncertain program completion time. The algorithm uses a probabilistic model of the cost of delay and the benefits of balancing to decide when to run a single balancing operation. The target applications of our load balancing algorithm have more deterministic behavior and our algorithm takes advantage of this fact, for example, it can run multiple balancing operations if they are needed.

Another key component of a load balancing algorithm is a redistribution algorithm which efficiently balances the workload. Biagioni and Prins [BP90] describe a data-parallel redistribution algorithm for grid-based computations which preserves the spatial locality of the workload it balances. Nicol [Nic92] describes a data-parallel redistribution algorithm which minimizes the volume of data transferred to achieve an optimal balance. These efforts are of interest because they describe data-parallel algorithms for redistributing load. In this paper, a data-parallel redistribution algorithm is described which balances a workload in which multiple independent tasks are associated with the same read-only problem data. Also, each processor stores only a minimal fixed amount of problem data. This reduces the memory space needed by the program during execution. The redistribution algorithm distributes a workload with these characteristics in a near-optimal manner. These characteristics are different from the characteristics of workloads manipulated by the other data-parallel redistribution algorithms.

In Section 2, the type of programs and workloads the load balancing algorithm works with is presented along with a high-level description of the load balancing algorithm. Section 2 also presents when and how load balancing occurs. Section 3, describes a data-parallel polygon rendering program to which the load balancing algorithm has been applied. Section 4 presents a performance study of the original and a load-balanced version of the rendering program. Section 5 presents conclusions.

2 The Load Balancing Algorithm

The load balancing algorithm uses a data-parallel programming model. In a data-parallel programming model, instructions are applied in parallel to each element of a data array. For the following discussion, we assume a virtual processing facility provides the abstraction of having one virtual processor assigned to each data element of a parallel array.

The load balancing algorithm can be applied to programs which compute the solutions of a collection of independent tasks. The tasks are independent in time; they do not have to execute in any specific order and there are no data dependencies between the tasks. Multiple tasks can share the same read-only problem data. An example of how multiple tasks share the

same problem data occurs in the polygon rendering application. Multiple tasks are used to process a row of pixels. Each task computes the solution for one pixel in the row. The pixel's location is computed by adding an offset to the row's initial pixel location. The initial pixel location is stored as part of the problem data.

In a prototypical program, each processor is assigned problem data and its associated tasks. A processor's workload is the number of tasks associated with its assigned problem data. Values in the range $1..number\_of\_tasks$ are used as indices to refer to these tasks. A task's index is used to calculate the specific problem data on which the task computes. In the pixel processing example above, the task index is used as an offset; task i processes the ith pixel location. The problem data and workload are stored in parallel arrays named *problem_data* and *workload*.

In order to process the tasks, the program increments a global task index counter, *index*, which starts at 1 and ends at the maximum workload of all the processors. During each iteration of the global index, each processor checks if they have a task with that index, and if they do, they compute a solution for the task. The instructions used to compute the solution of a multiple tasks in parallel are called the **solution phase**. A phase is a conceptual grouping of instructions in a parallel program which performs a useful activity. A pseudo-code description of a prototypical program is shown in Figure 1. Note that the WHERE statement activates processors for which the test is true and idles processors elsewhere.

```
<initial instructions>
forloop index = 1, MAX(workload)
   WHERE (index <= workload)
      Solution Phase(index)
   ENDWHERE
endloop
<further instructions>
```

Figure 1: A prototypical program to be augmented with the load balancing algorithm

As the computation proceeds, more and more processors complete the processing of their tasks and remain idle for the rest of the loop iterations. These processors are then termed *idle* processors. Processors which have tasks to complete are termed *active* processors. Processors are idled because all processors must process tasks with the same task index at the same time.

To improve the program's performance, the program is modified so that tasks with different

indices can be processed at the same time. The load balancing algorithm then distributes tasks from heavily loaded active processors to idle processors and tries to balance the workload among all processors.

How a program is augmented with the load balancing algorithm is now described. The load balancing algorithm consists of three distinct phases: the **information gathering phase**, the **decision phase** which decides when load balancing should occur and the **redistribution phase** which distributes tasks from active processors to idle processors. The basic iteration structure of the program is preserved. At the beginning of each iteration, the information gathering phase is executed. Then the decision phase is run, utilizing the gathered information to decide if balancing should occur during this iteration. If the decision is to balance, the redistribution phase is run, moving problem data from active to idle processors and assigning these idle processors new task indices to process. When tasks are distributed, the task indices originally assigned to an active processor can be assigned to multiple idle processors. Thus, different processors can work on different task indices during the same iteration. In the pixel processing example, this could mean, for example, that the first 3 pixels of one processor's row are processed along with the first 5 pixels of another processor's row. A parallel array, named *parallel_index*, is used to keep track of the current task index computed by each processor. A pseudo-code description of a prototypical load-balanced program is shown in Figure 2.

```
<initial instructions>
loop
   Information Gather Phase
   IF (Decision Phase returns TRUE) THEN
      Redistribution Phase
   ENDIF
   WHERE (workload > 0)
      Solution Phase(parallel_index)
      workload = workload - 1
   ENDWHERE
until (All elements of workload = 0)
<further instructions>
```

Figure 2: A prototypical load-balanced program

Note that the execution time of both the original and load-balanced prototypical programs

presented in Figures 1 and 2 is proportional to the maximum number of tasks assigned to any processor. The major difference between the programs, is the load-balanced program can reduce the maximum number of tasks on any processor by redistributing the workload.

2.1 When to load balance

The information gathering phase creates information which is used by the decision phase to decide if redistributing load will be cost-effective. This information includes a trial workload, named *new_workload*. This new workload contains a more balanced distribution of tasks on the processors. From this new workload a measure of the possible savings is calculated. The new workload has a smaller maximum number of tasks on any processor than the original workload. Thus, the savings are calculated as the maximum number of tasks on any processor in the original workload minus the maximum number of tasks on any processor in the new workload, as shown in the equation below:

$$\text{The } \textit{savings} \text{ in iterations} = MAX(workload) - MAX(new\_workload)$$

The savings are measured in terms of the number of future iterations of the loop which will execute the solution phase. The maximum number of tasks in a workload dictates the number of iterations that must be executed to process these tasks. If the new distribution is used then these "saved" iterations will not have to be executed.

The costs of the load balancing algorithm are also measured. Since the savings are measured in terms of the number of iterations, the costs are converted to this unit as well. The costs of the load balancing algorithm are incurred during the execution of the information gathering phase and redistribution phase. In order to quantify the costs of these phases, during each iteration their execution times are measured. The execution time of the solution phase during each iteration is also measured. $time_{info}$, $time_{redis}$ and $time_{soln}$ are the execution times of one execution of the information gathering, redistribution and solution phases. An estimate of the load balancing cost in terms of number of iterations can then be calculated by multiplying the sum of the execution time of the information gathering and redistribution phases by the inverse of the execution time of the solution phase, as shown in the equation below:

$$\text{The } \textit{costs} \text{ in iterations} = (time_{info} + time_{redis}) \times \frac{1 \text{ iteration}}{time_{soln}}$$

In order to provide a guarantee that the load balancing algorithm will always make cost-effective load balancing decisions, this cost measure must not be underestimated. Initial runs

of the load-balanced program on various datasets are used to compute an overestimated cost measurement. The longest information gathering time of any iteration and redistribution time of any iteration are then divided by the shortest solution time of any iteration for each dataset. The largest of the resulting cost measures provides an estimate of an upper bound on the load balancing cost in terms of iterations. To this initial estimate a constant is added to assure the cost measure will always be an overestimate. This overestimate is then used in all future runs of the load balanced program.

Utilizing the overestimated costs and the calculated savings a cost-effective load balancing decision is then made by the decision phase. If the savings are greater than the costs then the redistribution phase is executed. Since each load balancing decision results in a cost-effective iteration, the sum of these decisions results in a cost-effective program execution.

2.2 Reducing the variability of the cost measurement

It is important to reduce the variability of the execution time of the load balancing and solution phases. If there is a significant amount of variability, the cost overestimate will be extremely large and this can prohibit load balancing from occurring.

In order to reduce the variability of the load balancing algorithm, it uses communication routines which have predictable execution whenever possible. Two types of communication routines are involved: **general send and scan routines**.

A general send routine transfers data elements from one set of array locations to a different set of array locations. The execution time of a send routine depends on the number of data elements that are transferred and the locations they are transferred to. The redistribution phase utilizes a general send to redistribute the workload and problem data. It minimizes the use of the general send by transferring only a minimal amount of data and then copies this data to other processors with a scan operation.

A scan is parallel prefix operator; it successively applies an operator to elements of a parallel array creating a partial result for each element [Ble89]. For example, a scan addition operator stores the sum of the first i elements in the ith array location. Scans provide a useful mechanism for computing global sums, maximums and counts of the elements of a parallel array. Scan operations have predictable execution times because they use a fixed communication pattern. All the load balancing phases employ scans whenever possible.

The data-parallel programming model also helps to reduce the variability of the execution time because of the synchronous nature of the available programming constructs. Thus, timings are more predictable than they would be in an asynchronous message passing programming

model. An area of future work is to study how significant the predictability of the data-parallel programming model is to the effectiveness of this load balancing method.

Another area of concern is how to reduce the execution time variability of the solution phase. In particular, it is important to try to bound the minimum execution time of the solution phase because very short executions have the effect of greatly increasing the cost overestimate. For example, this problem can occur when a conditional in the solution phase causes a major portion of the instructions to be skipped during some iterations. One workaround for this problem is to introduce a short delay to provide some minimum execution time for the phase. This creates a lower bound on the execution time of the solution phase which in turn helps to place an upper bound on the cost overestimate. One disadvantage of this workaround is it increases the total execution time of the program.

2.3 How to load balance

Deciding when to balance is one important aspect of a load balancing algorithm. An equally important aspect is how balancing is performed. An efficient redistribution method is essential for good performance. In the load balancing algorithm, workload is distributed by copying the data from heavily loaded active processors to idle processors. The workload, in the form of task indices, is then divided up and assigned to the active and idle processors with copies of the problem data. Each active processor's workload is assigned some number of idle processors. This assignment is computed by assigning the idle processors in proportion to the workload on each active processor. Thus, heavy workloads are assigned more idle processors than light workloads and load is balanced evenly.

The following simple example illustrates the sequence of steps used to compute this assignment. It describes the essential core of the information gathering phase. In this example, seven processors are used. A sequence of numbers is used to represent a data-parallel array on these processors, with the first value in the sequence on the first processor, the second value on the second processor, and so on.

- **The following parallel arrays are used in the example.**

  Notice the problem data array only provides space for one element of problem data per processor. This means an idle processor can only be assigned to the workload of one active processor.

$$
\begin{array}{lcccccccc}
workload & = & 100 & 19 & 0 & 0 & 0 & 0 & 0 \\
parallel\_index & = & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
problem\_data & = & 1.0 & 2.0 & 0 & 0 & 0 & 0 & 0 \\
processor\_number & = & 1 & 2 & 3 & 4 & 5 & 6 & 7
\end{array}
$$

1. **Count the number of idle processors; those where the workload is 0.**

$$num\_idle = 5$$

2. **Compute the average workload of the processors and mask out all the active processors with workloads equal to or below the average.**

   The average is the lower bound on the maximum number of tasks there will be on any processor in the new workload after load redistribution. Therefore, workloads equal to or below the average are not assigned idle processors, since our goal when balancing is to reduce the maximum value in the new workload to as close to the average as possible.

$$
\begin{array}{lcccccccc}
avg & = & 17 & & & & & & \\
act\_mask & = & T & T & F & F & F & F & F
\end{array}
$$

3. **Compute the sum of the active iterations using the mask defined in the previous step.**

$$sum\_workload = 119$$

4. **To create the assignment, a proportion is computed for each masked processor. This proportion is the workload on each masked processor divided by the sum of the workloads on all the masked processors. This proportion is then multiplied by the number of idle processors and the result is truncated to an integer value. One is added to represent the participation of the masked processors.**

$$assignment = \frac{workload}{sum\_workload} \times num\_idle + 1$$

$$
\begin{array}{lcccccccc}
 & = & \frac{100}{119} \times 5 + 1 & \frac{19}{119} \times 5 + 1 & 0 & 0 & 0 & 0 & 0 \\
 & = & 4 + 1 & 0 + 1 & 0 & 0 & 0 & 0 & 0 \\
 & = & 5 & 1 & 0 & 0 & 0 & 0 & 0
\end{array}
$$

5. **The values in the new workload are calculated by dividing the original workload by the assignment array.**

$$new\_workload\_values = \frac{workload}{assignment}$$

$$= \quad 20 \quad 19 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0$$

The *new_workload* is instantiated by the redistribution phase if the decision phase decides balancing will be cost-effective. Its value is presented below for clarity. Notice the 100 task indices assigned to the first processor in the original workload will be assigned to first five processors in the new workload. The 19 task indices assigned to second processor in the original workload will be assigned to the sixth processor in the workload. This movement of tasks occurs because the redistribution algorithm uses contiguous blocks of processors to process the same problem data. More information on the redistribution phase is presented in Section 2.3.1.

$$new\_workload \quad = \quad 20 \quad 20 \quad 20 \quad 20 \quad 20 \quad 19 \quad 0$$

Using floating point computations and a close variant of the above steps[1] creates an optimal floating point based assignment of task indices to processors since all idle processors are assigned (possibly as fractions) and the number of task indices assigned to each processor are equal. This floating point based assignment cannot be used because fractions of processors cannot be assigned. The integer-based steps described above are used instead. These steps result in a near optimal solution. This near optimal solution can cause some idle processors to remain unassigned after a computation. For example, processor 7 remains unassigned in the example above.

The savings that can be achieved by utilizing the new workload are calculated by subtracting the maximum of the original workload from the maximum of the new workload values. In this example, the maximum of the original workload is 100 and the maximum of the new workload value is 20 resulting in a savings of 80 iterations.

[1]The above steps have been optimized for integer computations. In the floating point variant, the sum computed in step 3 totals all active iterations. In step 4, the assignment is equal to the workload over the sum computed in step 3 times the total number of processors.

$$assignment = \frac{workload}{sum\_workload} \times num\_procs$$

2.3.1 How to redistribute workload

If the decision phase indicates balancing is profitable, then the redistribution phase is executed. The redistribution phase sends the problem data from the active processors to the idle processors and updates the workload and parallel index arrays. This section summarizes the steps in the redistribution phase, building on the example started in the previous section.

The redistribution phase creates blocks of processors which process the same problem data. The size of these blocks is defined by the assignment array. The blocks are placed one after another from left to right in the new workload array. In the example, the first block consists of processors 1 through 5 and the second block consists of processor 6.

1. **An array of pointers to the first processor in each block is created by applying a scan addition operation to the assignment array using the** *act_mask* **defined in the previous section.**[2] **One is added to generate 1-based pointers. The array provides pointers to the processors where data should be distributed to.**

$$\begin{array}{lcccccccc} pointers & = & 0 & 5 & 0 & 0 & 0 & 0 & 0 \\ & = & 1 & 6 & 0 & 0 & 0 & 0 & 0 \\ processor\_number & = & 1 & 2 & 3 & 4 & 5 & 6 & 7 \end{array}$$

2. **The values in** *new_workload_values*, *parallel_index* **and** *problem_data* **arrays are sent to the beginning of each block using a general send operation guided by the** *pointers*. **The** *new_workload* **array is set equal to the** *new_workload_values* **array.**

$$\begin{array}{lcccccccc} new\_workload\_values & = & 20 & 0 & 0 & 0 & 0 & 19 & 0 \\ parallel\_index & = & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ problem\_data & = & 1.0 & 0 & 0 & 0 & 0 & 2.0 & 0 \end{array}$$

3. **The values at the beginning of each block in the** *new_workload*, *parallel_index* **and** *problem_data* **arrays are copied to all processors in their block using a segmented scan copy operation. A segmented scan operation applies a scan operation within each segment. The segments are set to correspond to the processor blocks.**

$$\begin{array}{lcccccccc} new\_workload & = & 20 & 20 & 20 & 20 & 20 & 19 & 0 \\ parallel\_index & = & 1 & 1 & 1 & 1 & 1 & 1 & 0 \\ problem\_data & = & 1.0 & 1.0 & 1.0 & 1.0 & 1.0 & 2.0 & 0 \end{array}$$

[2] We are using a variant of the scan addition operation which stores the sum of the first i elements in the *(i+1)*th array location and stores an initial value (in this case 0) in location 1.

4. **The *parallel_index* array is updated by creating a segmented scan addition[3] of the *new_workload* array and adding it to the index.**

$$\begin{array}{lcccccccc} scan\_add\_new\_workload & = & 0 & 20 & 40 & 60 & 80 & 0 & 0 \\ parallel\_index & = & 1 & 21 & 41 & 61 & 81 & 1 & 0 \end{array}$$

The first processor will solve task indices 1-20, the second will solve task indices 21-40, and so on.

This completes the description of the load balancing algorithm.

3 Application to a Polygon Renderer

The load balancing algorithm has been added to a data-parallel polygon renderer [OHA93]. The goal of a polygon renderer is to generate an image from a three-dimensional polygon model. A lighting model is used to approximate how light is reflected from the polygon model to an image which represents the user's viewpoint.

The purpose of the data-parallel renderer is to render extremely large polygon datasets at interactive rates on a massively parallel machine. These datasets are generated from scientific simulations and typically exceed 250,000 polygons. One motivation for rendering these datasets on the massively parallel machine is to avoid the high network cost of transferring the datasets to a graphics workstation. The steps taken by the rendering program include:

- *transforming*, which modifies the coordinates of the polygons to account for a new view position and magnification
- *clipping*, which eliminates polygons that are not in view
- *scan conversion*, which maps the polygons onto the rows of the resulting image
- *z-buffering*, which maps the scan lines generated in the previous step into pixels of the image

The scan conversion and z-buffering steps were initially implemented in the original renderer using data-parallel loops of the form shown in Figure 1. In the load balanced version of the renderer the steps were implemented using data-parallel loops of the form shown in Figure 2. For the scan conversion loop, the problem data is polygons and each task consists of creating and processing a scan line from a polygon. The number of scan lines in a polygon is dependent

[3]The same variant of the scan addition operation used in Step 1 on page 11 is also used here.

upon its image-space height. In the z-buffering loop, the problem data is scan lines and each task consists of creating and processing pixels from a scan line. The number of pixels in a scan line is dependent upon its length.

Unbalanced polygon datasets occur in a variety of ways. One way is during the creation of the polygon dataset from the output of scientific simulations. Isosurface algorithms are used to create polygonal representations of these outputs. The representation may be regularly or irregularly structured depending on the output and the isosurface algorithm used. The viewing angle and magnification can also cause imbalances. Clipping unbalances a dataset by eliminating tasks on processors which are assigned a polygon which is out of view.

4 A Performance Study

A series of experiments were executed using the original and a load-balanced version of the data-parallel polygon renderer. For all the experiments, the renderer generates output images which are 512×512 pixels in size. The experiments were executed on the Advanced Computing Laboratory's 1024 processor CM-5 at Los Alamos National Laboratory.

In the first experiment, performance data for the original and load balanced renderers is presented. The polygon datasets used in this experiment are a balanced and unbalanced version of the same scientific output, a hydro-dynamics simulation of an oil well perforator. They are generated by using two different isosurface algorithms. Table 1 provides the maximum and average workloads of these datasets. The first column of the table, lists the viewing and magnification transformations that have been applied to datasets. The "M" before the viewing angle means the dataset has been magnified. Notice in the balanced dataset the difference between the maximum and average workload is small, whereas in the unbalanced dataset the difference is large.

On the unbalanced datasets, the decision phase and the redistribution phase work together to effectively to improve the renderer's performance. Table 2 shows the results of rendering the unbalanced polygons with (**LB**) and without (**OR**) the assistance of the load balancing algorithm on 32, 64, 128, 256 and 512 processors of the CM-5. Notice the poor performance of the original renderer on the unbalanced datasets and the improvement obtained when using the load balancing algorithm. The performance of the load-balanced renderer provides a factor of 8 to 33 improvement over the performance of the original renderer on the unbalanced datasets.

Table 3 shows the results of rendering the balanced polygons with and without the assistance of the load balancing algorithm on 32, 64, 128, 256 and 512 processors. Notice that when the load balanced renderer is applied to the balanced datasets its performance is approximately

	Balanced			
	Scan		Z-Buffer	
View	Max	Avg	Max	Avg
(0,0)	8	5	7	1
(45,45)	10	5	8	1
M(0,0)	13	5	17	2
M(45,45)	21	1	21	3

	Unbalanced			
	Scan		Z-Buffer	
View	Max	Avg	Max	Avg
(0,0)	231	6	9	1
(45,45)	169	6	30	1
M(0,0)	512	8	14	1
M(45,45)	468	4	119	2

Table 1: The Maximum and Average Workloads of the Balanced and Unbalanced Datasets

	Number of Processors and Program Used									
	32		64		128		256		512	
View	OR	LB	OR	LB	OR	LB	OR	LB	OR	LB
(0,0)	48.91	6.21	28.98	3.57	18.18	2.12	12.76	1.35	9.99	0.92
(45,45)	38.03	5.33	22.37	3.00	14.27	1.78	10.45	1.15	8.24	0.81
M (0,0)	109.64	10.62	65.50	5.90	41.74	3.40	30.12	2.16	23.92	1.46
M (45,45)	99.13	4.42	57.91	2.42	37.19	1.46	27.33	0.94	22.40	0.67

Table 2: Rendering of Unbalanced Datasets in Seconds

the same as the original renderer. It is difficult for a load balancing algorithm to provide good performance on a balanced dataset since any redistribution steps will simply waste time. The empirical worst case performance loss is only 27 percent on balanced datasets when using the load balancing algorithm.

	Number of Processors and Program Used									
	32		64		128		256		512	
View	OR	LB	OR	LB	OR	LB	OR	LB	OR	LB
(0,0)	4.71	5.16	2.67	2.95	1.57	1.72	0.88	1.00	0.54	0.64
(45,45)	5.49	6.12	2.98	3.33	1.69	1.89	0.99	1.13	0.62	0.73
M (0,0)	10.61	12.47	5.80	6.77	3.18	3.73	1.82	2.21	1.11	1.40
M (45,45)	12.27	13.43	6.69	7.34	3.74	4.12	2.24	2.50	1.51	1.65

Table 3: Rendering of Balanced Datasets in Seconds

The original renderer's performance on the balanced datasets provides an estimate of the target performance we would like to achieve with the addition of a load balancing algorithm. The performance of the load-balanced renderer on the unbalanced datasets is within 70 percent of the performance of the original renderer on the balanced datasets.

In a second experiment, three other polygon datasets were tested. Two datasets were generated from different outputs of a fluid-dynamics simulation and the other from the output of a particle interaction simulation. Two of the the datasets are balanced and one is unbalanced. In summary, performance improvements ranged from a factor of 4 to 33. The empirical worst case performance loss is only 25 percent on balanced datasets when using the load balancing algorithm.

4.1 Cost Measurement Experiments

Overestimated costs of the load balancing algorithm were calculated for the unbalanced dataset shown in Table 2, by finding the sum of maximum execution times of the information gathering and redistribution phases and dividing them by the minimum execution time of the solution phase. Then a constant was added to ensure these values always overestimate the actual cost. The overestimated costs are 20 iterations for the scan conversion loop and 32 iterations for the z-buffering loop. It is instructive to relate these values to the maximum and average workload in Table 1. Note that the difference between the maximum and average iteration values on

many datasets exceeds these overestimates, implying that the load balancing algorithm may be able to improve the renderer's performance on these datasets. The overestimated cost values calculated for the unbalanced dataset were then used when executing the load-balanced renderer on all the other datasets. Cost values were computed for all the other datasets by dividing the maximum information gathering and redistribution phases execution times by the minimum execution time of the solution phase. These cost values were all less than the overestimates. In addition, the load balancing algorithm outputs a warning message for the user if the actual costs of any iteration exceeds the overestimated cost value.

The variability of the measured execution times of the load balancing phases of the load balancing algorithm are summarized in Table 4. Notice the information gathering and redistribution phases have little variability relative to the variability of the solution phase. The execution time of the solution phase of the scan conversion loop varies widely because it bypasses a major portion of the solution phase if there is no work to do. The execution time of the solution phase of the z-buffering loop varies because it utilizes an unstructured global send operation which transfers a variable amount of data. Even though this variability causes the costs overestimate to be large, substantial performance improvements are still obtained.

	Scan			Z-Buffer		
	Info.	Redis.	Soln.	Info.	Redis.	Soln.
Minimum	0.0002	0.0584	0.0098	0.0011	0.0592	0.0030
Maximum	0.0064	0.0703	0.3101	0.0093	0.0704	0.0458
Average	0.0012	0.0646	0.0812	0.0014	0.0641	0.0059

Table 4: Execution Time of Load Balancing Phases on 512 Processors in Seconds

5 Conclusions

A significant problem when using a load balancing algorithm is the possibility that along with improving performance on some datasets it will degrade performance on others. In this paper, a data-parallel load balancing algorithm was described which will not substantially degrade a program's performance on any dataset. This property results from utilizing an empirical measurement of the cost of load balancing along with a calculation of the possible savings to restrict load balancing to only when it is cost-effective. A data-parallel redistribution algorithm was described which redistributes a workload in a near-optimal manner. A performance study

of a polygon rendering program augmented with the load balancing algorithm was presented. The study showed the effectiveness of the algorithm, providing a performance improvement of a factor of 33 on unbalanced datasets and a maximum performance loss of only 27 percent on balanced datasets.

6 Acknowledgments

This research was performed at the Advanced Computing Laboratory of Los Alamos National Laboratory, Los Alamos, NM 87545.

References

[Ble89] G. E. Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[BP90] E. S. Biagioni and J. F. Prins. Scan directed load balancing for highly parallel mesh-connected parallel computers. In *Unstructured Scientific Computation on Scalable Multiprocessors*, pages 371–95, October 1990.

[Nic92] D. M. Nicol. Communication efficient global load balancing. In *Proceedings of the Scalable High Performance Computing Conference*, pages 292–299, April 1992.

[NJ90] D. M. Nicol and P. F. Reynolds Jr. Optimal dynamic remapping of data parallel computations. *IEEE Transactions on Computers*, 39(2):206–219, February 1990.

[NT89] D. M. Nicol and J. C. Townsend. Accurate modeling of parallel scientific computation. In *Proceedings of the 1989 SIGMETRICS Conference*, pages 165–170, May 1989.

[OHA93] F. A. Ortega, C. D. Hansen, and J. P. Ahrens. Fast data parallel polygon rendering. In *Proceedings of Supercomputing '93*, pages 709–718, November 1993.

[WPG91] M. C. Wikstrom, G. M. Prabhu, and J. L. Gustafson. Myths of load balancing. In *Parallel Computing '91*, pages 531–549, 1991.