

UNIVERSITY OF CALIFORNIA, SAN DIEGO

A Cache-Friendly Liquid Load Balancer

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science in

Computer Science

by

Federico David Sacerdoti

Committee in charge:

Professor Scott B. Baden, Chair
Professor Larry Carter
Professor Dean Tullsen

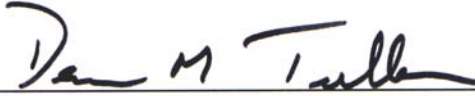
2002

Copyright

Federico David Sacerdoti, 2002

All rights reserved

The thesis of Federico David Sacerdoti is approved:







Chair

University of California, San Diego

2002

Table of Contents

Signature Page	iii
Table of Contents	iv
Table of Figures	vi
Vita.....	vii
Abstract.....	viii
I. Introduction.....	1
A. Parallel Machines and Clusters.....	3
B. Caches	4
C. Cache Friendliness	6
D. Load Balancing	9
E. Overview of Thesis.....	11
II. Cache Friendliness.....	13
A. Motivation.....	13
B. Existing Solutions	15
C. Design.....	17
1. Our Testing Application	17
2. Tiling for Cache	20
3. Autoblocker	21
4. Padding for Cache.....	23
5. EucPad2D	27
6. GcdPad3D.....	29
7. Pad.....	30
8. Virtual Memory Behavior.....	32
D. Experiment.....	33
1. Experimental Setup.....	34

	2. EucPad2D Experiments	35
	3. GcdPad3D Experiments.....	39
	E. Discussion	42
III.	Load Balancing.....	46
	A. Motivation.....	46
	B. Existing Solutions	48
	C. Design.....	51
	1. Problem Quantization	52
	2. Timings and the Autobalancer.....	53
	3. Space-Filling Curve	54
	4. Dampener.....	56
	D. Experiment.....	57
	1. Q-Type Experiments.....	58
	2. Uniform Balancer Experiments	60
	3. Non-Uniform Balancer Experiments	62
	4. Balancer Overhead.....	66
	5. Irregular Balance Experiments	67
	E. Discussion	69
IV.	Conclusion	73
	A. Future Work.....	74
	Appendix A: Tabular Data.....	77
	Appendix B: Users Guide.....	81
	References.....	87

Table of Figures

1. Non-Linear Node Performance.....	14
2. The RedBlack3D Kernel.....	18
3. The RedBlack3D 7-point Stencil.....	19
4. Tiled RedBlack3D Kernel.....	20
5. Auto-Blocked RedBlack3D Kernel.....	22
6. Figure of an Auto-Blocked Array.....	22
7. Padding and its effect on Memory Layout.....	24
8. Cache Conflict between Adjacent Columns.....	27
9. The EucPad2D Algorithm.....	28
10. The GcdPad3D Algorithm.....	30
11. Pad vs. GcdPad.....	31
12. RedBlack2D Grind Time vs. N: EucPad2D.....	35
13. RedBlack3D Grind Time vs. N: EucPad2D.....	37
14. RedBlack3D Fortran vs. Auto Blocking.....	39
15. RedBlack3D Grind Time vs. N: GcdPad3D.....	40
16. RedBlack3D Grind Time vs. N: GcdPad3D with 16 Procs.....	41
17. The Hilbert Space-Filling Curve in 2D.....	55
18. Quantization Experiment with 8 Procs.....	59
19. Balancer at Rest.....	61
20. Non-Uniform RedBlack: Unbalanced.....	63
21. Non-Uniform RedBlack: Balanced.....	63
22. Load Blancer in Action: Balance Efficiency.....	64
23. Balancer Performance.....	65
24. Balancer Overhead.....	66
25. Balancer Pseudocode.....	70

VITA

1976	Born, Washington D.C.
1998	B.S. Washington University, Saint Louis
1999	Founder, Sacerdoti Linux Machines, Saint Louis
2000-2002	Research Assistant, University of California, San Diego
2002	M.S., University of California, San Diego

FIELDS OF STUDY

Major Field: Engineering

Studies in Computer Engineering.

Professors George Varghese, Kenneth Wong, Washington University, Saint Louis

Studies in High Performance Parallel Computing.

Professor Scott B Baden, University of California, San Diego

Studies in Computer Networking and Distributed Systems.

Professors George Varghese, Keith Marzullo, University of California, San Diego

ABSTRACT OF THE THESIS

A Cache-Friendly Liquid Load Balancer

by

Federico David Sacerdoti

Master of Science in Computer Science

University of California, San Diego, 2002

Professor Scott B. Baden, Chair

Experience with computing clusters indicates a load balancer is necessary to allow programs to adapt to a machine as they run. Scientific applications are partitioned into pieces, which run concurrently on nodes of parallel computing clusters. Simple methods of problem partitioning yield static-sized portions for each node. A dynamic load balancer allows the problem to resize itself on a parallel cluster so that idle nodes receive more work as appropriate, essentially allowing a problem to flow through different parts of the machine, aiding efficiency and improving performance.

Our work in this thesis shows that when designing a parallel load balancer, further issues must be considered. Local performance on compute nodes has an important impact on balancing that is often overlooked. We show that by minimizing cache conflict misses on nodes, we can achieve more accurate load balance.

To address cache conflicts, we present a two-level partition scheme for parallel applications. We show how the use of cache tiling and padding techniques within our two-level framework improves performance and yields more reliable performance metrics, which are essential for our dynamic load balancer.

This thesis describes our dynamic, cache-friendly tiling method, and associated liquid load balancer design. We explore several recent cache-tiling algorithms, optimized for 3D stencil-based codes. Our load balancer then uses these tactics to obtain reliable per-node performance data. We expect this approach to yield a highly adapted computing environment for scientific solvers and simulators on modern heterogeneous clusters, leading to significant speedups for a wide range of applications.

Chapter I

Introduction

The needs of scientific computations have outstripped the available resources for as long as computers have existed. As a result, a good deal of effort has focused on increasing the efficiency of high performance computer systems. The parallel computer has emerged as the platform of choice for high performance applications. By harnessing multiple computing elements that work together in concert, parallel systems can provide state-of-the-art performance for scientific computations. Load Balancing is a much-studied strategy for improving the efficiency of parallel computers. This technique seeks to reduce processor idleness during program execution by carefully allocating computation in the system. Existing methods of load balancing have largely overlooked the behavior of processor caches, however, which we address in this work. This thesis presents our Cache-Friendly Load Balancer for scientific cluster machines.

Load balancing involves assigning parts of a parallel application, called its *partitions*, to computing elements, called *nodes*, in such a way that all the nodes have just the right amount of work to keep themselves busy. Balancing is necessary because many parallel programs concentrate on certain areas of the computation more than others. If we view the problem as having a *size*, such as in a Finite-Element-Analysis of the stresses on a spaceship, we would split the ship into pieces, and call each one a *partition* of the problem. We assign partitions to various nodes of the cluster in some way to ensure all nodes receive equal work. If each ship partition required equal computation, this

assignment would be easy. However, if certain parts of the ship require more effort to compute than others, balancing the problem on the machine becomes a difficult task.

The ability to load balance effectively depends on our capacity to detect differences in the workload distribution across processors. There are several methods available. If we determine the workload of each partition at the beginning of the problem by some analytical means, we could assign partitions to avoid any imbalance. However, predicting application performance is difficult in general. In addition, hardware differences between cluster systems make predictions not only application-specific, but machine-specific as well. A second choice involves measuring application performance at each node *during* computation. We choose this method because its success does not depend on an analytic cost model of performance. Once node workload is determined, we can move partitions between nodes during runtime to improve load balance.

Our work in this thesis concentrates on insuring single-processor performance over a range of parameters. We show that partition sizes interact with the processor cache on a node, leading to a large impact on its performance. By making the parallel application cache-friendly, we can mitigate these effects, affording us predictable performance over various partitioning parameters and problem sizes. This in turn allows us to achieve high performance and accurate load balancing together.

The contribution of this work is a two-level work-granularity framework that allows a load balancer to choose partition sizes without risk of degraded performance. Our two-level tiling framework removes the interaction between partition size and performance. We implement this framework in a run-time library that can be made to adapt to particular hardware footprints more easily than compiler-based schemes [21]. As we will show, cache-friendliness depends heavily on hardware architecture and partition size.

A. Parallel Machines and Clusters

Parallel computers are the system of choice among the high-performance scientific community due to their speed and low cost/performance. These types of machines are composed of many processing elements called nodes, which can run autonomously to perform tasks in parallel. By splitting a problem among the nodes of the system using a process called *partitioning*, an n -node parallel computer can theoretically finish an application n times sooner than could a single processor. The general strategy when building such a computer is to use the fastest processors possible for each node, and the fastest interconnect between nodes.

In the 1970s and 1980s, the fastest processors were in the vector-type computers pioneered by Seymour Cray and Cray Research. At that time, although Personal Computer (PC) microprocessors existed, their performance was several orders of magnitude slower than the advanced vector CPUs. These processors also cost several orders of magnitude more than their PC cousins, limiting the size of parallel computers of that era. During the 1990s, however, PC microprocessor technology began to close the performance gap between themselves and the vector processors. The PC manufacturers achieved this improvement by adopting some of the technology developed earlier by Cray, such as interleaved memory, high clock speeds, and large register files. In addition, better CMOS processes, pipelining technology, and widespread use of caches helped improve PC processor performance as well.

With fast and cheap microprocessors available, larger parallel computers were built that could outperform their vector counterparts for many types of applications. Since computer vendors often sold their generic systems with a few number of CPUs each, parallel systems evolved as a cluster of these machines, connected with a fast interconnect network. A single vendor typically built these systems, and they were homogeneous in the sense that all nodes had the same hardware. The Blue Horizon

cluster [29] is a parallel machine of this type. This system has many nodes connected by a high-bandwidth switch; each node has 8 high-performance workstation processors made by IBM-Motorola. As single-processor systems became faster, however, the parallel computer landscape changed again.

The latest trend in clusters involves using commodity, off-the-shelf PC processors and components for each node. Since these nodes are inexpensive, commodity clusters often outperform homogeneous clusters due to their sheer size. Scientists can afford more nodes for the same price. These clusters rely on cheap but high performance CPUs, which recently have become available. One result of this design is heterogeneity within the cluster.

Computer vendors do not generally build commodity clusters. They are the first type of parallel computer often built by researchers themselves, using components from different manufacturers. The Beowulf project¹ popularized this type of machine, and its utility prompted similar efforts, including the Rocks clustering project at SDSC [27] whose Meteor cluster we use in our experiments. Components in commodity clusters exhibit a wide range of designs and architectures, to the point where a cluster built over the period of a few months may contain nodes of different specifications. Although this heterogeneity only has important implications for our load balancer if node cache sizes are different, our model has the ability to handle such variations.

B. Caches

All modern processors have a memory *cache*, a French-derived word for “store”. In parallel clusters, each CPU on a node has its own private cache system. The idea behind cache is to use a small amount of fast but expensive memory to hide the long access times to the large, slow, (but inexpensive) main memory. Luckily, a simple model for

¹ <http://www.beowulf.org/>

cache layout can achieve this goal because of two important attributes of computer programs. Statistical studies have shown that most programs have a high degree of both spatial and temporal locality with respect to the memory they use [13, 15, 16, Ch 5]. Temporal locality occurs when a memory location used will likely be used again in the near future (a few thousand clock cycles on the CPU). Spatial locality, a related property, happens when the application uses memory locations corresponding to nearby points in physical problem space. For example, if we access data located in address x , we will likely access data in locations $x+1$, $x-1$ in the near future.

The simple cache layout model therefore places a small amount of fast memory in between the CPU and the Main Memory. Currently we make this cache using SRAM technology, with an access time of around 2-10 CPU clock cycles per memory access. The cache fills with data from main memory, and due to spatial and temporal locality it generally contains most data the CPU requires. Increasing the likelihood that the cache will contain a given CPU memory request is of central importance to this thesis. This figure is called the *cache hit rate* and we define it as the fraction (*cache hits / memory requests*), where a cache hit means a data request is present in the cache, and the processor does not need to obtain it from main memory.

In the last few decades, caches have become more and more important. CPU clock cycles have increased faster than memory read-write latency [16, fig 5.1]. As a result, processors must wait longer to obtain data from memory, and this trend has remained constant for the last decade or so². In the late 1980s, a CPU could expect a memory request to return with data in around 10 clock cycles, even if did not exist in cache. A current PC processor must wait much longer, on the order of 100 clock cycles, to receive data from main memory. Since effective caching can reduce this time to around 2-3

² Processor Speeds double roughly every 18 months, while memory speeds only increase around 7% per year.

cycles even for the fastest processors, an important effect of this trend is that cache hit rates become more critical to performance.

We will present our work on improving cache effectiveness in a later chapter, but in order to explain why traditional parallel computations are generally cache *unfriendly*, we must describe cache operation in some detail here.

C. Cache Friendliness

When a cache does not contain a requested piece of data, the processor has to wait hundreds of cycles to obtain the word from main memory. In this section we show how by being careful we can reduce the number of these *cache misses* during program execution.

If all memory requests were found in cache, the cache hit rate would be 100%. Since the cache has a smaller capacity than main memory, this ideal is unachievable. The CPU must experience some cache misses, which are classified into three types [14]. *Compulsory* misses happen when a piece of data is requested for the very first time; since data only comes into cache when it is requested, the first request must cause a miss. Although there exist hardware strategies to avoid compulsory misses, such as large cache block sizes and pre-fetching [25, 16, § 5.3], we do not attempt to mitigate them with software techniques.

The second and third types of cache miss are of primary concern to us. *Capacity* misses occur when the cache is full and cannot accommodate more data without discarding some current resident. Often, the application will request the discarded data again in the future, due to locality principles, and the cache will have to re-load it from main memory. The cache is most efficient when we reuse its contents frequently. Imagine Michelangelo painting the Sistine chapel. He has setup his scaffolding to access a particular area of the ceiling within his reach. He could choose to paint one meter of

ceiling at a time, like a raster-scan, from the northernmost wall of the church to the southernmost wall. However, it is much more efficient for Michelangelo to paint one figure at a time. By doing so, he reuses his scaffolding placement as much as possible. Similarly, to maximize cache reuse we would like to concentrate on a portion of the application at once, such that each portion fits into cache.

Our cache-friendly strategies address capacity misses by computing a small *block* of the problem at a time. This technique, called blocking or *tiling* [48, 49], chooses each portion so it fits into cache without spilling over [17]. During the computation of a block, the CPU will not experience any capacity misses since the block fits into cache.

With compulsory misses unavoidable, and capacity misses removed via blocking, *conflict* misses remain. This last class is the most subtle, architecture-dependant, and difficult to ameliorate of the three types. The *many-to-one mapping problem* makes solving conflict misses difficult. Since a cache is smaller than main memory, its hardware designer must decide where to place data. Imagine a 1,048,576-location main memory (2^{20} addresses) and a 256-location cache (2^8 addresses). Since the cache must be able to store any one of the million pieces of memory, we need a flexible mapping scheme that chooses a cache location from a memory location. A direct map is the simplest solution, popular with earlier CPU designs. In this strategy, the last 8 binary digits of the memory location specify which of the 256 cache locations to choose, a perfect fit.

The insight with direct mapping is that the cache may evict an existing piece of data d before the cache is completely filled, causing an unnecessary miss if the CPU requests d in the near future. We consider this *conflict miss* unnecessary since the cache was not full when it discarded d , the discard happened simply due to a conflict in the mapping. In our example, a conflict arises if two desired pieces of memory happen to reside 256 locations away from each other in memory. To address this issue, designers have altered the cache hardware to allow a small number of memory addresses to map to the same cache

location without eviction. This mechanism, called *n-way set associativity*, helps avoid conflict misses, and is generally reasonably effective. High degrees of set-associativity, however, require more complex hardware and slow down cache operation.

Due to the increased latency of highly set-associative cache hardware, many processors implement a multi-level caching strategy. The highest and fastest level of cache (level 1 or L1) often has a very low associativity, making it fast but susceptible to conflict misses. Larger and slower caches in the system (L2, L3) typically have higher set-associativity, and are less prone to cache conflicts. The Power3 processor by IBM [47] has a very high 128-way set-associative data cache. However, the processor's high cost and relatively low performance discourages its use in commodity parallel clusters.

Previous work [7] has shown that cache conflict misses hamper performance on single-processor machines with direct-mapped L1 and L2 caches. Our results show on modern processors that employ set-associative cache designs, conflict misses continue to affect performance. Our single-processor and cluster systems have 16-way set-associative L2 caches, which can avoid the conflict misses generated by our application. To achieve the highest clock speed, however, the L1 cache in both systems is only 2-way set-associative and prone to conflicts. Our experiments show that L1 conflict misses have an adverse effect on performance, even on advanced processors. (See the Experiment section in Chapter 2 for full details.)

To reduce cache conflict misses, we work around the limitations of the cache mapping function. By carefully choosing which memory locations hold our data elements, we can influence how they will be mapped into cache. We do this in such a way that no element will overwrite another in cache until it has filled to capacity. We describe the techniques fully in Chapter 2.

D. Load Balancing

Load Balancing attempts to assign problem partitions to compute nodes in a cluster in such a way to evenly distribute computational workload. We can see the utility of balancing a parallel problem by noting that we consider its running time to be that of the last node to complete the computation. If we do not distribute the problem carefully, some processors may lay idle while others work. The unloaded nodes do not contribute their full capacity towards the computation, increasing the running time of the application. Although achieving a good balance may be trivial for simple problems, non-uniform applications make load balancing a difficult problem since we cannot easily determine the workload of a partition anytime before its execution.

We can view an unbalanced computation as having different work *pressures* in different parts of the parallel machine [32]. Overloaded nodes have a high pressure, while idle nodes experience low pressure. A parallel load balancer avoids this pressure imbalance, either by partitioning the problem correctly at the outset (static balancing), or by detecting pressure at runtime and transferring work between nodes to alleviate it (dynamic balancing).

Static load balancers partition the (generally static) problem once, and with enough care that the application remains balanced throughout its execution. This type of balancer typically attempts to minimize the amount of communication between nodes, partitioning the problem where the fewest number of data dependencies exist, while assigning each partition the same number of data elements [50]. We show in Chapter 3, however, that if partitions have different sizes, their compute time per point, or *Grind Time*, varies due to differences in the cache miss rate. Dissimilar grind times lead to a natural load imbalance, since although the partitions contain nearly the same number of elements, they take different times to compute. Our two-level tiling framework removes the

interaction between partition size and performance. Therefore, the technique equalizes the compute time of the partitions, leading to better load balance for static problems.

Dynamic load balancing algorithms apply to a more general set of problems than their static counterparts, and have received a great deal of attention in the field, as the survey in [33] attests. Advanced load balancers [1,8,10] follow the dynamic design, and allow chunks of work to move in a fluid manner through the machine, settling so the work pressure is even. Although recent studies have explored the quality of various dynamic load balancing methods based on communication characteristics [52], our work concentrates on the importance of considering single-node performance in the balancer design, which is a less-studied aspect of the problem.

We begin by showing how load balancing can increase the number of conflict misses in cache. A dynamic balancer will transfer work between nodes in the machine, as necessary, to alleviate load pressures. Imagine the balancer decides to move work to an unloaded node u from another node in the system. A naïve balancing approach would simply add the incoming data elements to the local partition on u . As shown in the next chapter, changing the size of the partition can affect performance unpredictably due to cache conflict misses. Furthermore, this performance effect is non-linear and difficult for the load balancer to predict. Therefore the balancer's decision to move work is tenuous: it does not know how data motion to node u will affect its performance after the transfer.

Moreover, the load balancer would like to assume all nodes are created equal. If this were true, a partition would have the same computational weight on every node. Some commodity clusters are heterogeneous, so this assumption may not be valid. A partition that runs quickly on one node may take longer to compute on a different node due to hardware differences. These hardware variations can also lead to increased cache conflicts that further affect performance.

The balancer we present in Chapter 3 mediates poor performance. It addresses the problem of changing partition sizes by imposing a level of granularity to the problem, and moving only fixed sized problem units called *quanta*. Think of quanta as small problem partitions, sufficiently numerous that each node receives many of them. The balancer addresses the problem of different cache conflict rates by optimizing each quantum specifically for the local cache. In this way quanta will always be cache-friendly, even if nodes in the cluster have different cache characteristics.

Two characteristic levels of granularity are employed in the balancer design. The top level consists of the problem quanta. We then tile each quantum individually, to be cache-friendly. The second level of granularity ensures that quantum size does not adversely affect performance. We must use two levels in the design because the ideal tile size for each node may not correspond to the size of its quanta. A node can own several quanta with different shapes, but it requires a rectangular, constant-sized tile for maximum local performance.

To reduce communication between nodes, our balancer uses a Hilbert Space-Filling curve [1, 53, 54] to assign problem quanta to processors. The load balancer uses a simple geometric RCB algorithm [44] to partition the workload along the 1D Hilbert curve. Full details of our balancer design are given in Chapter 3.

E. Overview of Thesis

In this thesis we establish that cache-friendliness is an important property for parallel computations, especially with respect to load balancing. We present several methods of improving cache-friendliness, and then show how it helps the efficiency of our load balancer.

This thesis is organized as follows. Chapter 2 presents our cache capacity and conflict miss strategy, and its associated experimental results. Chapter 3 describes our two-level

blocking strategy for load balancing, including the design and experiments of our dynamic liquid load balancer using space-filling curves. Chapter 4 summarizes our work and discusses opportunities for future study.

Appendix A presents tabular data for all the graphs presented in this document. Appendix B describes the Autoblocker and Autobalancer APIs in the KeLP [23] framework, along with a short users guide to the balancer.

Chapter II

Cache Friendliness

A. Motivation

When optimizing parallel applications, single processor performance is often overlooked. Designers view the cluster as a whole, and typically model a parallel cluster as a graph of abstract *black-box* computing nodes, and the network that connects them. Most parallel software techniques concentrate on inter-node behavior, the interplay of one node with its neighbors. In this chapter we argue that the nodes themselves should be given a closer look.

Our experiments have shown that single-node performance, the computational efficiency of one node in isolation, is an important factor for parallel performance. In addition, measurements indicate that single-node performance varies with problem size and problem shape. We show in this chapter that by using techniques to reduce cache miss rates, we can achieve high performance independent of problem or partition size and shape. Although this result has been shown previously [17], we show the relevance of this problem to load balancing and partition-size choice in parallel applications.

Our dynamic load balancer design relies heavily on single-node load figures to decide how to move work through the cluster. When we measure the load on a given node, we essentially gauge its performance p as it computes a workload of problem elements, w . In initial experiments, we found that p is sensitive to small variations in the size of w . In fact, if we gave the node slightly more work, even for uniform problems, the performance p could vary significantly. Since our balancing algorithm anticipated a linear relationship between p and w , it occasionally made poor decisions that lead to more load imbalance.

As a result, we began to examine the factors affecting node performance, seeking to find a smoother workload-to-performance ratio.

The second motivation for smoothing the performance-to-workload ratio is for parallel applications with non-uniform partition sizes. Graph partitioners typically assign equal number of elements per partition [50]. However, partition *shapes* also affect the cache miss rate, as shown in this chapter. Therefore load imbalance can occur because the partitions of a parallel problem have unequal performance, despite their equal size. By applying cache-friendly techniques to each partition, we can decouple performance from partition shape as well as size.

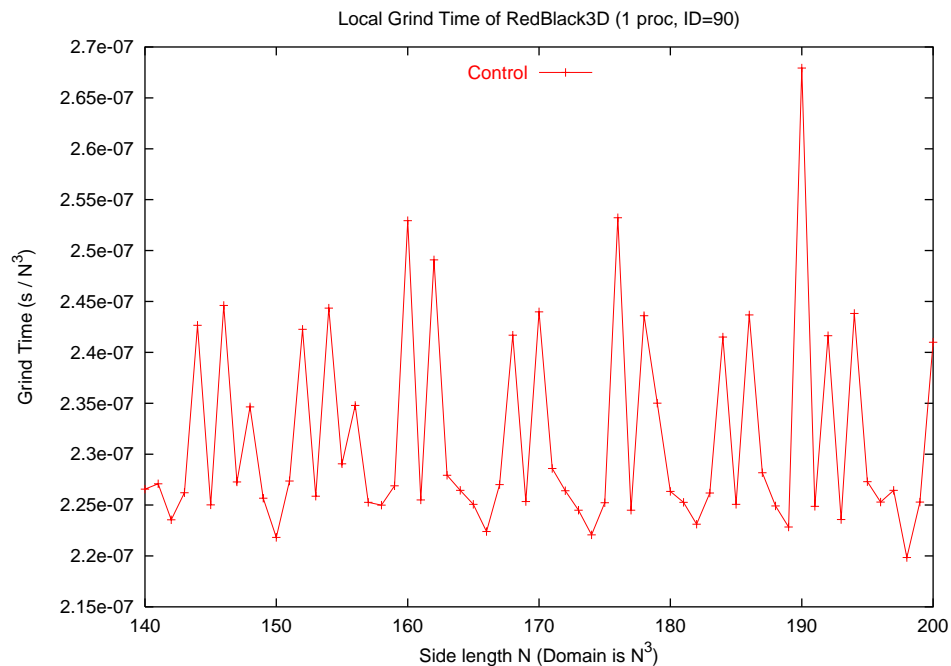


Figure 1. Inconsistent Performance with Changing Problem Size. An example of the kind of performance behavior we would like to avoid. This experiment shows the *Grind Time* (time needed to compute a single element) of a RedBlack3D PDE solver for different values of N , the side length. No tiling or padding techniques have been applied for this run, and we note the grind time varies by up to 20%. The experiment was performed with the environment described in the Experiment section, on the Alpha machine.

As discussed in the introduction, a node's processor cache has a large impact on its performance due to the gap between CPU and memory speeds. Therefore, we looked to the cache effects on a node to explain the non-linear work/performance ratio. Previous work reveals that an undesirable work array size can dramatically increase the number of conflict misses in cache [5]. We show that cache miss rates vary with the stride length between memory accesses, a function of the partition's size and shape. Frequent cache misses can increase running times drastically. As seen in Figure 1, performance can vary 20% over a small range of workload sizes. Although the array shape in this experiment is always cubic, the memory access strides increase with the array size. We hypothesize that the performance fluctuation seen in Figure 1 is due to a varying cache miss rate caused by changing memory access strides. In the experimental section we validate this hypothesis.

Using runtime techniques to improve the cache miss rate, we achieved smooth performance under a range of partition sizes. By effectively decoupling the partition shape from node performance, we put load balancing decisions on firmer ground. In the process, we were also able to achieve a performance speedup for most problems. In the next section, we present the existing work on caches, and the various techniques used to improve their effectiveness.

B. Existing Solutions

In the previous section we motivated our need to examine the factors that affect cache miss rates. Here we present past work that identifies and addresses these issues.

The first commercial machine built with a processor cache was the IBM 360/85 [19]. Smith [15] performed a seminal analysis of the importance of cache, its usefulness apparent even in these early systems. Hill [14] defined the three types of cache misses, Compulsory, Capacity, and Conflict, as discussed in Chapter 1. Splitting a computation

into small chunks, called *tiles* or *blocks*, to improve locality for cache was a known technique to improve the memory efficiency of numerical codes [48]. This strategy became widely adopted by the numerical computation community, as epitomized in the popular BLAS [20] linear algebra package. It was not until Lam, Rothberg, and Wolf's analysis of cache conflict misses [17], however, that these simple tiling schemes were reevaluated. They identified the need to tailor the cache tile size to the work array size, instead of using a fixed-sized tile based on the cache capacity. Newer tiling techniques compute tile size as a function of the work array shape to more efficiently reduce conflict misses in cache.

Recent work has focused on carefully choosing cache tile sizes based on the work array size. Coleman and McKinley [5] present an algorithm that chooses a tile size based on fitting columns of a work array A evenly into cache. Their algorithm ensures the first tile dimension evenly divides the cache capacity, a powerful technique that helps reduce conflict misses. Our approach hinges on ideas originally presented in this paper. Rivera and Tseng [4,6,7] build on this technique, and produce a simple function to quickly calculate tile sizes for 3D problems that minimize both capacity and conflict misses in cache.

Both Coleman and Rivera use numerical stencil codes in their analyses, similar to the RedBlack PDE solver that we chose as our motivating application for our load balancer (see next section). As Rivera and Tseng showed smooth performance over varying problem sizes as one of their results, we choose the EucPad2D algorithm in [5] as a starting point for our cache explorations. Although they intended their technique for use in compilers, we use it to dynamically select a custom tile size for each node in the cluster using a runtime library.

Mitchell et al. [22] include other levels of memory hierarchy account when choosing tile sizes, such as the processor’s TLB³ cache in addition to data and instruction caches. We use a simpler technique that only considers cache specifications. We do identify the occurrence of TLB misses in our 2D results, however, and the incorporation of the TLB cache into tile size selection is a technique we would like to adopt in future work.

In the next section we describe the theory and function of the tiling algorithms, addressing their approaches to capacity and conflict miss in cache.

C. Design

This section describes the design of our cache-friendly strategy for parallel applications. We begin by presenting the application we used for the theoretical analysis of the algorithms and our experiments. We then describe the theory of tiling, followed by various padding strategies. Finally, we analyze the memory overheads incurred from padding.

1. Our Testing Application

For our analysis and experiment, we used a Fortran numerical kernel called RedBlack. This kernel implements a simple Partial Differential Equation solver for Laplace’s equation $\nabla^2\phi = 0$ with Dirichlet boundary conditions, using the Gauss-Seidel RedBlack method [31]. We chose this particular example for its simplicity and low memory requirements. RedBlack falls in the category of stencil codes, a common style of numerical solver that makes successive passes over an array, updating each point as some function of its neighbors. RedBlack’s computation behavior is representative of an important class of finite-difference applications typically run on scientific parallel

³ The TLB, or Translation Lookaside Buffer, is a specialized piece of fast memory that keeps track of Virtual to Physical memory page mappings. It is vital for the fetching of any memory address.

clusters. In addition, RedBlack and its cousins are somewhat akin to the fruit fly *Drosophila melanogaster* in genetics: they are the standard test bed for new techniques. Coleman and McKinley [5], Mitchell et al.[22], and Rivera and Tseng [4,6,7] all use similar codes to develop their ideas.

Our version of RedBlack is implemented in Fortran77, and uses the KeLP framework [2,3,18,23] to coordinate partitioning and communication of the parallel application. KeLP adds a single layer of communication elements, called *ghost-cells*, around the surface of each partition of the problem. It uses these extra memory elements to store off-processor neighbor data that are updated each iteration using MPI messages. In this manner, each local RedBlack kernel obtains data from its neighboring nodes with a simple memory access to the ghost cells. The KeLP system ensures these cells are refreshed every iteration. Using ghost cells, KeLP effectively hides all communication activity from the application. Figure 2 describes the RedBlack 3D kernel in the Fortran language.

We implement our cache algorithms within this framework. By using KeLP, we have precise control over how the following RedBlack algorithm is called, which we use to our advantage for some of the techniques presented later in this section.

```

double precision A(a10:ah0,a11:ah1,a12:ah2)
double precision RHS(a10:ah0,a11:ah1,a12:ah2)

do k = a12+1, ah2-1
  do j = a11+1, ah1-1
    jk = mod(j+k,2)
    do i = a10+1+jk, ah0-1, 2
      A(i,j,k) = c *
2          ((A(i-1,j,k) + A(i+1,j,k)) + (A(i,j-1,k) +
3           A(i,j+1,k))) + (A(i,j,k+1) + A(i,j,k-1) -
4           RHS(i,j,k)))
    end do
  end do
end do

```

Figure 2. The RedBlack 3D Fortran kernel. Operates over the array A, the innermost i loop traverses down columns; the j loop, along rows; and the k

loop, over planes. The loops leave room for the single coating of ghost cells on all sides of A . The RHS array represents the right hand side of the Laplace equation.

For our analysis, the most important feature of the RedBlack application is its memory access pattern. For every point p in our 3D work array A defined by $p=A(i,j,k)$, RedBlack accesses p 's neighbors in each primary direction: $i\pm 1$, $j\pm 1$, and $k\pm 1$. Including the point itself, RedBlack reads 7 elements from memory. These accesses lie in the five-column *neighborhood set*, which includes the columns containing p and its neighbors. We call this pattern a *stencil*. Figure 3 shows the 7-point RedBlack stencil covering three planes and five columns of A . Memory accesses to the stencil are the primary cause for cache hits and misses during program execution.

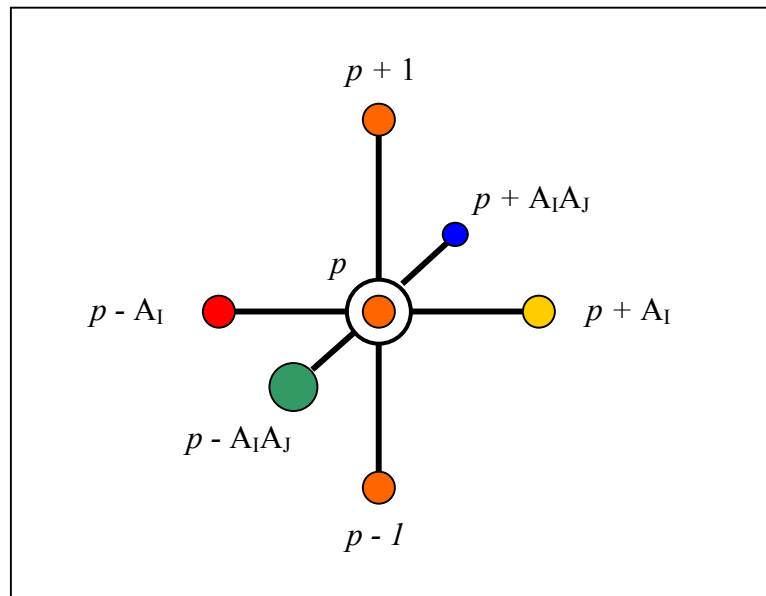


Figure 3. The RedBlack3D 7-point stencil. Each node above represents a memory location. This memory access pattern is used in the analysis of the cache tiling and padding algorithms. The center point p is the active point, while its neighbors shown in the diagram compose its *neighborhood set*. The node labels show how far from p the node is in memory, where A_1 is the column length, and A_j is the row length of the work array A .

Now that we have presented our example application, we discuss the two types of cache optimizations used to improve its performance predictability.

2. Tiling for Cache

Tiling for cache [56] is a well-known technique that allows more effective use of the computer memory hierarchy. If a work array A exceeds the cache capacity, tiling effectively partitions A into small chunks that can each fit entirely into cache. Due to spatial and temporal locality, RedBlack will likely reuse an element of A repeatedly during its execution. However if we do not tile, the cache may have to evict a given element *before it is ever reused*. With a correctly tiled program, the CPU will fill the cache once from memory, and then reuse the cached data multiple times. Since the processor can access cached elements more quickly, tiling yields a performance advantage. We note that tiling for cache is analogous in some sense to the high-level partitioning we have done among the nodes in the cluster.

```

double preciTion A(a10:ah0,a11:ah1,a12:ah2)
double preciTion RHS(a10:ah0,a11:ah1,a12:ah2)

do jj = a11+1, ah1-1, Tj
  do ii = a10+1, ah0-1, Ti
    do k = a12+1, ah2-1
      do j = jj, min(jj+Tj-1,ah1-1)
        jk = mod(j+k,2)
        do i = ii+jk, min(ii+jk+Ti-1,ah0-1), 2
          A(i,j,k) = c *
2          ((A(i-1,j,k) + A(i+1,j,k)) + (A(i,j-1,k) +
3           A(i,j+1,k)) + (A(i,j,k+1) + A(i,j,k-1) -
4           c2*RHS(i,j,k)))
          end do
        end do
      end do
    end do
  end do
end do

```

Figure 4. The tiled RedBlack 3D Fortran kernel. Tiling is implemented in the Fortran code with the two tile dimensions T_i and T_j , which are provided by the tiling algorithm.

Our tiling and padding algorithm chooses the tile dimensions T_I , T_J , and T_K so that the entire tile can fit into the local processor cache. A subtle point is that T_K , the number of tile planes that need to fit in cache, is only used to determine T_I and T_J . We do not actually tile along the Z axis of A. Examining the memory access pattern of RedBlack3D, only four⁴ planes need to be present in cache at once. Therefore, we do not care how many *total* planes are in the tile, only that any four of them fit concurrently in cache.

3. Autoblocker

In this section we will introduce a helpful feature we created called the *Autoblocker*. In our runtime library, tiling must be explicitly described in the code. We can do this directly in the Fortran kernel loops, as illustrated in Figure 4. The autoblocker is a feature of our runtime framework⁵ that coordinates tiling automatically. Applications using our system do not know they are being tiled, while still benefiting from the technique. A parallel algorithm such as the RedBlack3D kernel in figure 5 simply computes over one region (the red tile in figure 6), while initializing its arrays to a larger region (the full array A in figure 6). The autoblocker calls the algorithm once per tile, incurring a slight subroutine-startup overhead that is negligible in our results.

The autoblocker has several advantages over compiler-based tiling. Compilers require some runtime support for blocking, which the autoblocker supplies. In addition, standard Fortran compilers can be used while allowing the application to benefit from the latest tiling and padding techniques.

⁴ Three planes for the 7-point stencil, and one for the right hand side matrix of Laplace's equation $\nabla^2 \phi = \rho$, where ρ is the right hand side.

⁵ The Autoblocker is a C++ Iterator, is a special object that coordinates a loop. The Autoblocker coordinates a loop through the cache tiles. The Fortran kernel is called once per iteration by the Autoblocker.

```

double precision A(a10:ah0,a11:ah1,a12:ah2)
double precision RHS(a10:ah0,a11:ah1,a12:ah2)

do k = a12, ah2
  do j = TjLow, TjHi
    jk = mod(j+k,2)
    do i = TiLow+jk, TiHi, 2
      A(i,j,k) = c *
        2 ((A(i-1,j,k) + A(i+1,j,k)) + (A(i,j-1,k) +
        3 A(i,j+1,k)) + (A(i,j,k+1) + A(i,j,k-1) -
        4 c2*RHS(i,j,k)))
    end do
  end do
end do

```

Figure 5. Auto-Blocked RedBlack3D Fortran kernel. This kernel is called once per tile by the Autoblock C++ iterator. Both the extra tiling loops, and ghost cell allowances have been removed. The a indices used when initializing the A array are from the full array A , while the T indices used in the loops define the current tile.

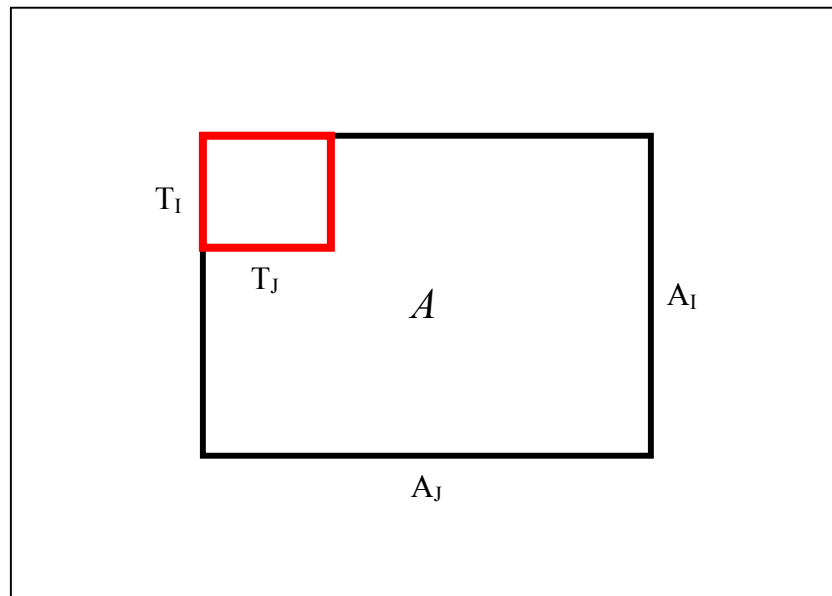


Figure 6. Front face of a 3D autoblocked array. The Autoblocker gives the Fortran RedBlack kernel the dimensions of the work array A ($A_I \times A_J$) to initialize its array, but instructs the kernel to only compute over the tile region defined by T_I and T_J .

4. Padding for Cache

Previous work has shown that conflict misses in cache make a significant contribution to poor performance in scientific applications [17]. Our results show this type of cache miss also causes timing irregularities with respect to local array size. Hardware remedies for conflict misses exist, principally highly set-associative cache design. This hardware approach, however, increases latency and so is often absent from the cache closest to the processor [26, 28]. Certain architectures, such as on the IBM Power3 processor [47], have high set-associativity in the L1 (first level) cache, and therefore will benefit less from these techniques.

As conflict misses play a big role in performance, and hardware counter-measures are often lacking in strength, our cache-friendly techniques focus on avoiding them with a smart layout of data in memory. This method, called padding, alters the data location in memory by placing blocks of dummy elements between the columns and planes of our local array A . Recall from the introduction that when we cache a word of data, its cache location is based on the last few bits of the word's memory address. Conflicts occur when two pieces of requested data, say d and e , share the same low-order address digits⁶. When this happens, the cache must evict one location (say d) since it is unable to place the other elsewhere. Cache hardware can use n -way set-associativity to allow a small number of data words (n) to map to the same location in cache without evicting its previous resident. However, the first-level processor caches used in our experiments are only 2-way set-associative [26,28]. If the work array A happens to be a particularly ill-shaped, our 3D stencil code could potentially place 5 often-reused words in a given cache location. Therefore relying on hardware alone to resolve cache conflicts is not enough.

⁶ Let the address of $d = 1000599$, and the address of $e = 2000599$. If the cache uses the final 3 address digits to locate the words, they will conflict in a direct-mapped cache. We use a decimal representation of memory addresses in our discussion for clarity.

Fortunately, since we know the cache mapping mechanism, we can arrange the columns and planes of our three-dimensional array A in memory such that their cache locations do not overlap. We begin with the observation that memory is laid out as a one-dimensional structure, the blue curve in figure 7. The compiler and operating system lay out our 3D array A in consecutive memory locations.

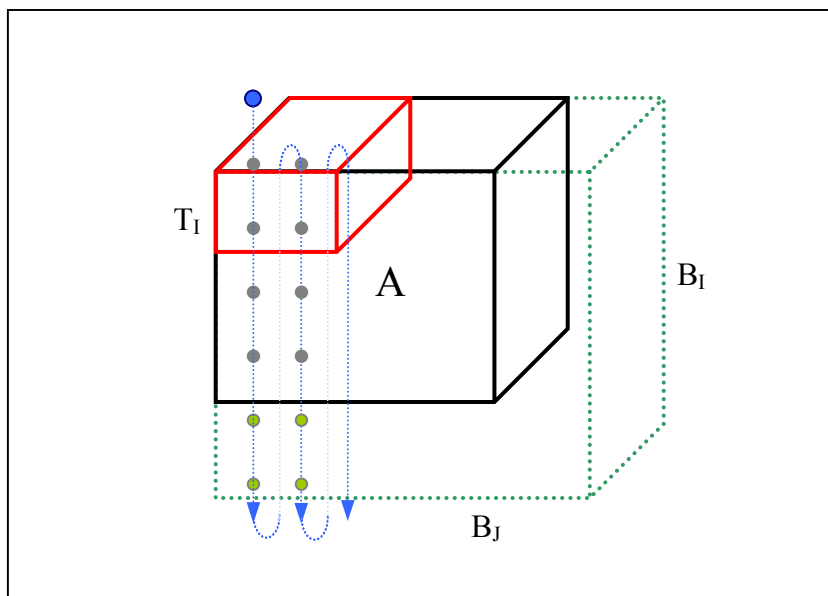


Figure 7. Padding and its effect on Memory Layout. The array A is tiled as in the previous figure. In addition, it has been padded on two axis, and now achieves the dimensions B_1 and B_j . The vertical lines and dots represent memory locations, which traverse the array like a 1D string. The padding alters the memory locations of the data in the second column by forcing the memory string to extend through the padded area. By carefully choosing the amount of padding, we can reduce conflict misses in cache. GcdPad3D pads in this manner.

Padding adds extra elements, unused by the application, between columns and planes in A . Although we do not use these extra locations, they take up space in memory, forcing our actual data to new locations. We have now changed the memory addresses of all columns in A except the first one. Since the memory location of our data has changed, so has the cache address for each element. We can use this mechanism to force the cache to place columns of A in favorable, non-conflicting locations. The reader should note this

method wastes memory, but for our purposes the performance advantages outweigh this shortfall. Later in this section we describe how the Virtual Memory system in Linux helps ameliorate this cost [51, § 22.6].

Our second observation is that RedBlack stencil code reuses many elements along a column, its primary axis⁷. We infer this locality property from inspection of our code’s central loop, but it is a common characteristic of many Fortran applications, which arrange their data in a column-major order. We therefore desire an entire column to remain in cache at once so the CPU can quickly access its elements. Note that since we have tiled the computation, the column size is that of the tile, not the entire array A .

We use the padding technique to ensure no two columns within a tile overlap in cache. In addition to this constraint, we want the columns of a tile T to be as long as possible, while still fitting our RedBlack’s five-column neighborhood set in cache. Longer columns lead to higher performance due to cache-filling speeds from memory⁸. To address these requirements, we explore a class of algorithms that choose the column length T_1 such that it is close to the *greatest common divisor* of the cache size and array column length [5,6,7].

$$T_1 = \text{gcd}(C, A_1) \tag{1}$$

When we choose our tile shape in this manner, the following lemma holds.

Lemma 1: For a direct mapped cache, every column in tile T will either overlap another column in cache completely, or not at all.

Proof: If two columns in a tile could overlap partially, then it must be possible for a column d to be split at the end of the cache C , where a

⁷ The primary axis is the one that iterates down consecutive memory locations. For column major ordering, this is the column axis, denoted in this document by $0 \leq i \leq I$.

⁸ Modern EDO RAM modules can read consecutive pages of memory quickly (their read setup time is a bottleneck that is not imposed on contiguous page reads), so the difference between filling a short column and a long column in cache is small.

prefix of d maps to the end of C and a suffix of d maps to the beginning of C . Since by (1) T_1 is a divisor of both C and A_1 , $T_1 \% C = 0$, and the suffix of d will always be of length 0. Therefore no such column exists and the lemma holds.

As odd array sizes will always lead to $T_1 = 1$, the actual cache algorithms relax the above equation by using a cost function to pick the most effective tile shape. We discuss the details of the cost function and algorithms in later sections. For this analysis, however, we assume that equation (1) strictly holds.

Since no column of a tile will partially overlap another in cache, we are assured that in a tile, the CPU will encounter no conflict misses, as long as no column in the neighborhood set interfere. However, by lemma 1 a tile column can *completely* overlap another. To see when this happens, we examine the active neighborhood set of columns. Remember that we define the neighborhood set as the five columns surrounding a given point p in the tile.

Consider a simple 2D case: the neighborhood set contains the p 's column, and the columns of its immediate neighbors to the left and right. If $C \mid A_1$, we will have a problem, since all adjacent columns will conflict⁹. Figure 8 illustrates this situation. The first column T_1 in the tile will map to the first T_1 locations in cache, the solid red line in the figure. A gap of $(A_1 - T_1)$ memory locations will follow before the next column T_2 begins, the dotted black line. However, since $C \mid A_1$, this means $T_1 + (A_1 - T_1) = kC$ for some k . Therefore T_2 , the dotted red line, will land directly on top of T_1 , and the cache will evict the entire column T_1 to make room.

This case is unlikely since if $C \mid A_1$, then $\gcd(C, A_1)$ must equal C , and we will choose $T_1 = C$. If $T_1 = C$ we could only fit one tile column in cache, which is not a good tile for RedBlack2D. In the 3D case, however, fully conflicting columns is a danger. For 3D problems, we need to ensure that $C \mid A_1 A_j$ is always false to prevent columns in adjacent

⁹ By $a \mid b$ we mean “ a evenly divides b ”. Equivalent to $b \% a = 0$.

planes from conflicting with each other. Equation 1 does not insure this property, and there is no way to detect when it may happen. Later we see that conflicting planes lead to poor performance.

In this analysis we assume a direct-mapped cache. Since our machines actually have a 2-way set-associative L1 cache, this may be acceptable in some situations. However, in our experimental section we show that performance suffers for certain problem sizes if we ignore complete conflicts between adjacent tile planes.

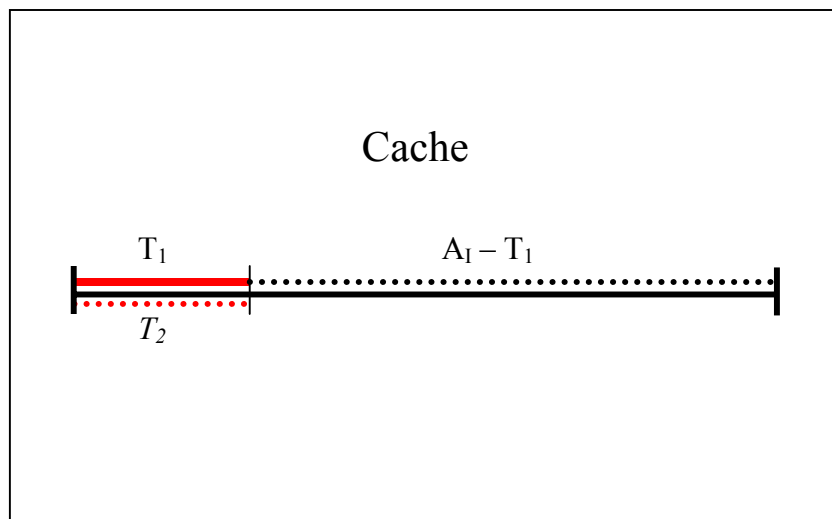


Figure 8. Cache conflict between columns in the neighborhood set. We assume a direct-mapped cache. If A_1 evenly divides the cache size C , then adjacent columns will conflict completely, even with tiles chosen by EucPad2D and GcdPad3D. This type of interference can occur between adjacent planes of A in problems padded using EucPad2D.

5. EucPad2D

We begin our padding efforts by examining the simpler 2D case. We use the EucPad2D algorithm from Tseng & Rivera [4,6]. This strategy chooses the tile column length, T_1 , to satisfy equation (1) so T_1 evenly divides both the work array A and the cache size C . The number of tile rows, T_j , is chosen so the tile fits into cache, meaning $T_1 T_j \leq C$. EucPad2D chooses the tile size that satisfies these criteria, and has the smallest

cost. The cost function is the same as used by Coleman and McKinley [5], and favors square tile shapes. Padding occurs by adding elements to the end of columns in A , changing their length by 0-8 elements. Each padded size yields viable tiles, and we choose the one with minimum cost. Coleman and McKinley indicated this cost function reduced cross-interference misses for a matrix-multiply kernel, but point out this cost metric may lead to TLB-cache misses. They note each column in the tile will often require its own TLB entry, since memory addresses in different columns are far apart. As TLB misses are expensive and stall the cache entirely, we should ensure the number of rows (T_J) does not exceed the number of TLB entries. This is less of a problem in 3D codes since their array dimensions are smaller, and adjacent columns may share a single TLB entry.

The running time of EucPad2D follows that of the $\text{gcd}(a,b)$ algorithm, which completes in $O(\log a)$ time, where $a > b$.

```

L = ∅
EucPad2D(H, Hnext, Wprev, W):
    L += {H,W}
    if Hnext != 0:
        Wnext = H/Hnext*W + Wprev
        EucPad2D(Hnext, H % Hnext, W, Wnext, Pad)

```

Figure 9. EucPad2D Pseudocode. The recursive function is called with the arguments $(C, Col, 0, 1)$, where C is the cache size, and Col is the column size of A that is potentially padded. L is the list of possible tile sizes. The algorithm essentially computes the $\text{gcd}(C, Col)$ and chooses it as the tile column size.

Originally, we used the simple EucPad2D algorithm to choose tiles for our RedBlack2D stencil. When experiments showed no speedup (see the discussion section), we used the algorithm to choose tiles for a 3D RedBlack application. We now saw a speedup, but the performance irregularity persisted. Although tiling reduced the capacity

misses, leading to a speedup, conflicts among planes in the neighborhood set hampered performance for certain problem sizes.

To tile A for a 3D RedBlack application, we insure that four full planes fit in cache. To this end, we instructed the 2D algorithm to assume the cache was one-fourth its actual size, leading to correspondingly smaller tiles. Unfortunately, this simple solution fails to consider conflicts between planes. The front and back columns in the neighborhood set will conflict for certain problem sizes. Our results show when tile planes conflict, performance suffers. Therefore this tiling technique yields limited benefits for 3D problems. The next tiling algorithm takes inter-plane conflicts into account, and leading to more predictable performance.

6. GcdPad3D

The GcdPad3D algorithm [7] pads both the columns and the planes of our 3D array. In addition to insuring (1), GcdPad3D also ensures a similar conflict property for tile *planes*.

$$T_j = \text{gcd}(C, A_j) \tag{2}$$

When we choose our tile shape in this manner, the following lemma holds. A tile plane is a slice of a tile, or a set of T_j tile columns.

Lemma 2: For a direct mapped cache, every *plane* in the tile T will either overlap another plane completely, or not at all.

Proof: Proof follows from Lemma 1's proof with the word *column* replaced by *plane*.

GcdPad3D pads our array A until both equations (1) and (2) are true, so neither tile columns nor entire planes will partially conflict with one another. Therefore GcdPad3D insures that columns from adjacent planes do not conflict, avoiding the weakness of the previous algorithm.

```

GcdPad3D(C, Ai, Aj, Ak) :
    Tk=4
    # Ti = the smallest power of 2 >= sqrt(C/Tk).
    Ti = pow(2, ceil(log2(sqrt(C/Tk)))
    Tj = C/Ti*Tk
    Bi = 2*Ti*floor( (Ai+3*Ti-1)/2*Ti ) - Ti
    Bj = 2*Tj*floor( (Aj+3*Tj-1)/2*Tj ) - Tj
    # Max pad of Ai should be + 2Ti-1 elements,
    # same for Aj.

```

Figure 10. GcdPad3D Pseudocode. The algorithm picks a favorable square tile shape based on the cache size C , then pads the array A until $T_I = \gcd(C, A_I)$ and $T_J = \gcd(C, A_J)$. Assumes that C is an even power of 2, $C=2^k$ for some k . The tile depth T_K is always fixed at 4 for the stencil code's access pattern. B_I and B_J are the new dimensions of A after padding.

The first phase of the algorithm chooses a desirable tile size based on the dimensions of A and the cache size C . It attempts to pick the largest square tile by choosing T_I and T_J as a function of \sqrt{C} . At this point it has not made any effort to insure that the tile is non-conflicting. In the second phase, the function pads the array A to ensure the tile meets this criterion. GcdPad's strategy is markedly different from EucPad2D in this respect. The latter picks the tile size based on the current size of A , while GcdPad3D chooses the tile size first, and then coerces the array to a desirable size such that equations (1) and (2) hold. The maximum padding possible to A is $(2T_I - 1)(2T_J - 1)$. The actual impact of this padding on the application's memory footprint is less, however, due to characteristics of the virtual memory system.

As the GcdPad3D algorithm involves only a series of arithmetic operations, it completes in $O(1)$ time.

7. Pad

As GcdPad incurs a memory overhead, it is natural for us to search for another algorithm. The Pad algorithm [7] is purported to have nearly the same desirable

performance characteristics of GcdPad, while requiring less memory overhead from padding. A simulation of the Pad algorithm revealed, however, that for the 3D array size range 50^3 to 200^3 (typical of our problem quanta), Pad does not lead to a significant improvement over GcdPad in all cases. As seen in figure 11, Pad provides a significant advantage¹⁰ over GcdPad in 52% of the sizes in our range. When Pad reduces the memory overhead, it does so by 30-100%, however the benefit applies to about half the array sizes tested.

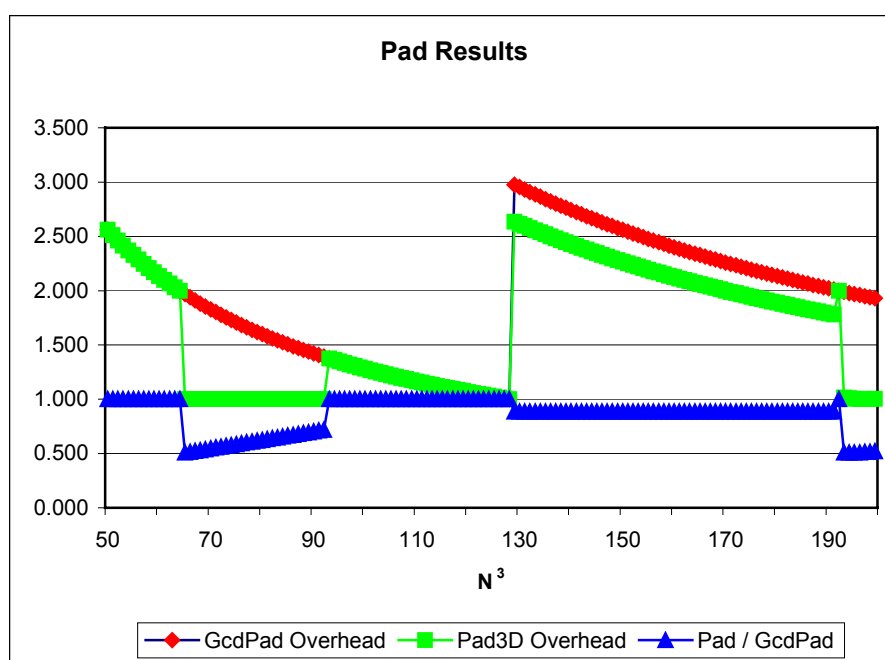


Figure 11. The comparison of GcdPad3D and Pad3D cache padding techniques. Values are from algorithm simulations with a range of 3D problem sizes (X-axis). Overheads represent physical memory overhead incurred by padding for cache. An overhead of 2.0 indicates the padded application requires twice as much physical memory as the unpadded version. Virtual Memory behavior is taken into account in this graph.

The added complexity of the Pad algorithm makes these gains less attractive. Pad runs in $O(N^2 \log C)$ time, where N is the side length of the problem, and C is the cache

¹⁰ We define significant as a 25% reduction in memory overhead.

size. The N^2 term comes from an exhaustive search through the $(B - A)$ domain, where B is the size of the padded array returned by `GcdPad`. `GcdPad`, however, takes only $O(1)$ time to complete. Since we must run `Pad` for each problem quanta (as they may have unique sizes), we feel in our design the increased running time of this algorithm does not justify its reduction in padding overhead.

One potential optimization to this `Pad` involves pre-computing a lookup table of results for likely quanta sizes. However, since quanta may not be cubic, the solution space is too large to make this practical. In the discussion section, we point to some hopeful leads for improving `Pad`'s running time.

In the next section we discuss how the operating system's virtual memory helps reduce the actual memory overhead from padding automatically. We believe that previous estimates of padding overhead [6, 7] overstated its cost.

8. Virtual Memory Behavior

In Linux as in most modern UNIX variants, new memory is not mapped to physical RAM until it is accessed. When we request new memory from the OS, it simply alters the heap break point (`sbrk(0)`) in the virtual memory address space. Essentially, it gives us a range of virtual addresses that do not occupy any real memory. Only when our program tries to access an element of a page¹¹ in this new memory does the OS take action to allocate physical memory [51, § 22.6], a strategy called demand-paging. Generally, code will access new memory soon after it allocates it, but this is not true for our padded regions. Certain pages of padding will never be accessed by the application, and therefore will never use physical memory. Specifically, since we organize our program data in column-major order, padding between planes of a 3D application occupies large contiguous areas of memory that typically span multiple pages.

¹¹ A page is a chunk of memory containing a few KB of data, 4KB on Linux.

To finish our argument that unmapped VM pages help reduce the cost of padding, we note that VM addresses are plentiful (3GB/process in Linux), and that the cache uses Virtual memory addresses for its location mapping, not physical ones. Therefore, we will achieve the desired cache location of our data even if the padding is never put into physical memory on the machine.

The caveat is that when we access *any* location in a VM page the *entire* page gets mapped to physical memory. So padding along the columns of A will still increase the memory overhead of the application (recall A is in column-major order). A discussion of the precise overhead incurred by GcdPad3D appears later in the chapter.

In this section we have presented the theory and design of our tiling and padding algorithms, with the goal of reducing capacity and conflict cache misses on the local nodes. In the next section, we describe our experiments with these algorithms, and discuss our results.

D. Experiment

This set of experiments show the results of applying our cache-friendly strategies to RedBlack problems of various sizes and dimensions. The arrays are made up of 64bit floating-point values, specified by the *double* type in the C/C++ language, and the *double precision* type in Fortran. To show the effects of our algorithms, we vary the size of the work array by increasing its side length N . All the 2D arrays are square, containing N^2 total elements and the 3D arrays are cubic with N^3 elements. We measure the Grind Time performance metric for each experiment.

The performance figure we present is the application Grind Time. This value represents the time needed to compute one element in our work array A , and provides a performance figure that is independent of problem size. We calculate Grind Time as (*Running time for one iteration / elements in A*). This metric almost measures time/flop of

the application, but will be an order of magnitude high since the RedBlack kernel performs approximately 10 floating-point operations on each element. The grind time gives an absolute performance figure for the experiment, which provides a good indication of the stability of the performance-to-workload function that motivated our consideration of caches.

1. Experimental Setup

Our experiments were performed on two hardware platforms. The first is a single processor machine we will refer to as the Beta configuration. Beta has a single AMD Athlon processor [26] running at 850Mhz. Since our cache algorithms target single node performance, running on the one-processor Beta machine allowed us to gather applicable results without using expensive supercomputing time.

Beta uses the Linux operating system with kernel 2.5.3, an experimental version similar to the stable 2.4.15 Linux release. Its software comes from the Mandrake 8.1 distribution, including the gcc compiler version 2.96 that was used to compile the C++ and Fortran experimental code. Beta is equipped with 256MB SDRAM, and its Athlon Thunderbird processor has a 64KB, 2-way set-associative L1 data cache, a 256KB full-speed 16-way L2 cache, and a 32-entry data TLB cache. Its dedicated system bus to memory runs at 200Mhz. See [26] for more details on the Athlon processor.

The second machine we use is the 200-node Rocks [27] Meteor parallel cluster at SDSC. In this section, we refer to this machine as the Gamma configuration. All nodes in Gamma run the Linux kernel version 2.4.9, and use RedHat 7.2 software, including gcc version 2.96 that we used to build the executables. Gamma is a heterogeneous cluster, but most nodes are 2-way Intel Pentium III servers running at 1Ghz, with 512MB SDRAM shared between both processors. The Pentium III has a 16KB, 2-way set-associative L1 data cache, a 256KB, full-speed 8-way L2 cache, and a 64-entry TLB. The processor uses

a dedicated 133Mhz system bus to main memory. See [28] for more details on the Pentium III processor.

2. EucPad2D Experiments

The first experiment ran on a 2D problem. We use the EucPad2D to choose a tile size, and potentially pad the column size of our array A by 0-8 elements, depending on its side length N.

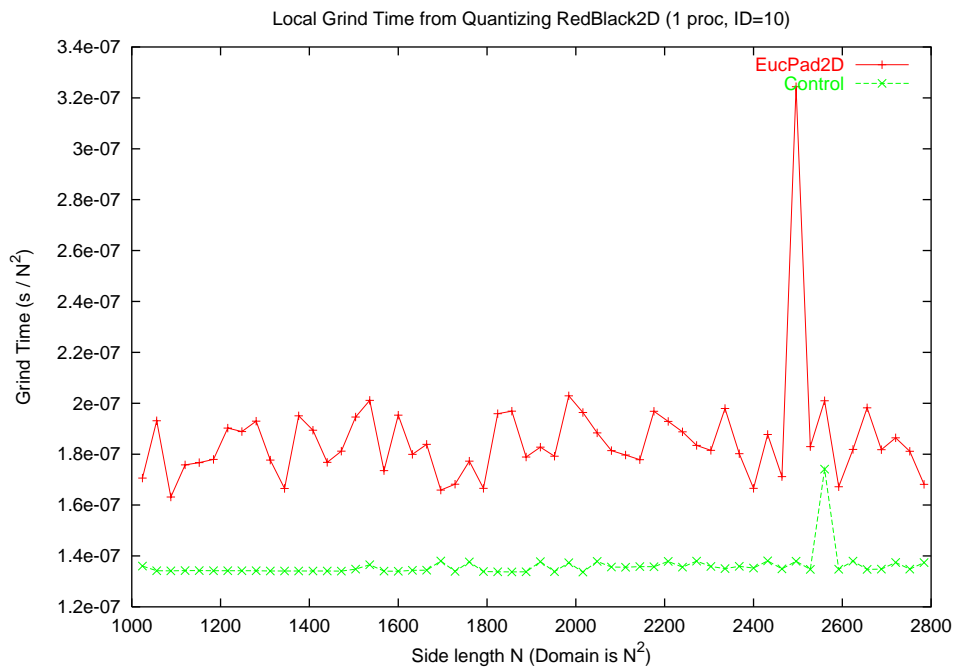


Figure 12. Grind Time plot from Tiling and Padding a 2D version of the RedBlack application on the single-processor machine called Beta. The simple EucPad2D algorithm was used to choose tile sizes, and tiling was implemented in the Fortran kernel (Fortran-blocked). The word *Local* in the title indicates that only the numerical kernel run time was measured.

We actually see a slowdown from tiling in this experiment, due to several factors. The first reason is that hardware prefetching in the CPU can help ensure the cache remains full during the entire execution [25], by detecting the simple memory access pattern required by RedBlack2D and anticipating its future requirements. RedBlack2D

has a 5-point stencil that iterates over three columns in a regular pattern, so the next iteration it will ask for elements +2 memory locations away from each current element. The CPU's prefetcher picks up on this pattern easily, and predictively fills the cache with those elements. It will fail at the end of every column, since the +2 stride does not hold, but this only results in $O(A_I)$ cache misses. As we had expected our application to generate many more misses, our tiling performance suffers.

The second reason for the slowdown is due to TLB misses. As described in the Design section, each column in a tile may require its own TLB entry, causing misses when accessing the work array A. To make matters worse, the RedBlack kernel requires another array to be present in cache, the Right Hand Side (rhs in the code) for the computation of each element. Since this array has the same size as A, but lies in a different location in memory, it requires another TLB entry per tile column.

In Linux, the page size is 4KB, and Beta's Athlon CPU has 32 TLB entries. In the $N=2560$ run, we gave each tile 27 columns, and each column has 15 elements. Therefore columns are spaced over 20KB away from each other in memory¹². Therefore every column does indeed require its own TLB cache entry.

RedBlack2D accesses 3 columns of A and one of RHS for each point. The processor's 32 TLB entries will fill after the 8th column in the tile ($32/4$). All subsequent columns will cause two TLB misses, one for the new column of A, and one for the new column of RHS. In our example we will see $(27-8)*2 = 38$ TLB misses for every tile. Misses in the TLB are expensive, and will stall the cache completely, blocking the processors pipeline until the correct VM-to-Physical memory page mapping has been read from memory. Mitchell et al. described this problem in [22]; they suggest making the tiles "thinner" to reduce the number of TLB entries used per tile.

¹² Distance: $64\text{bits/element} * 2560\text{ elements/column} = 163,840\text{ bits or }20,480\text{ bytes}$.

The control experiment runs more efficiently, since without tiling, it uses a page completely before moving to the next one. Therefore, it uses each TLB entry to its fullest extent, while the tiled version uses only T_1 elements of each page before skipping to the next tile column (15 in our example). Because of their penalties, TLB misses are another cause of the performance degradation we see in the early experiments.

The next graphs show the affects of the EucPad2D algorithm when applied to a 3D problem. As described in the Design section, our 3D adaptation of EucPad2D has the flaw that for certain problem sizes, every adjacent column in a tile will conflict. This property will cause some performance variations in the experiment.

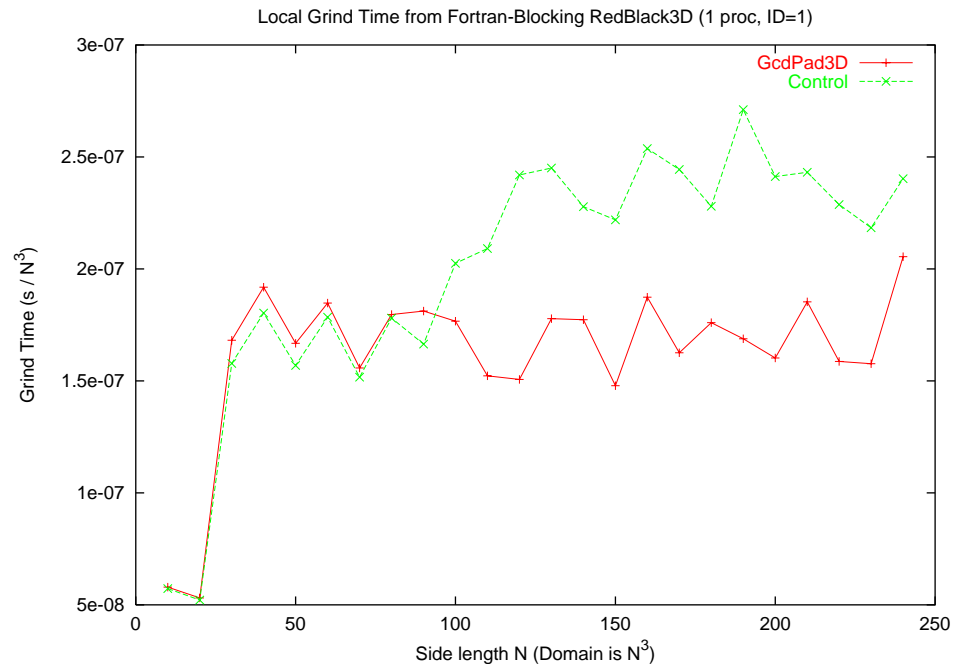


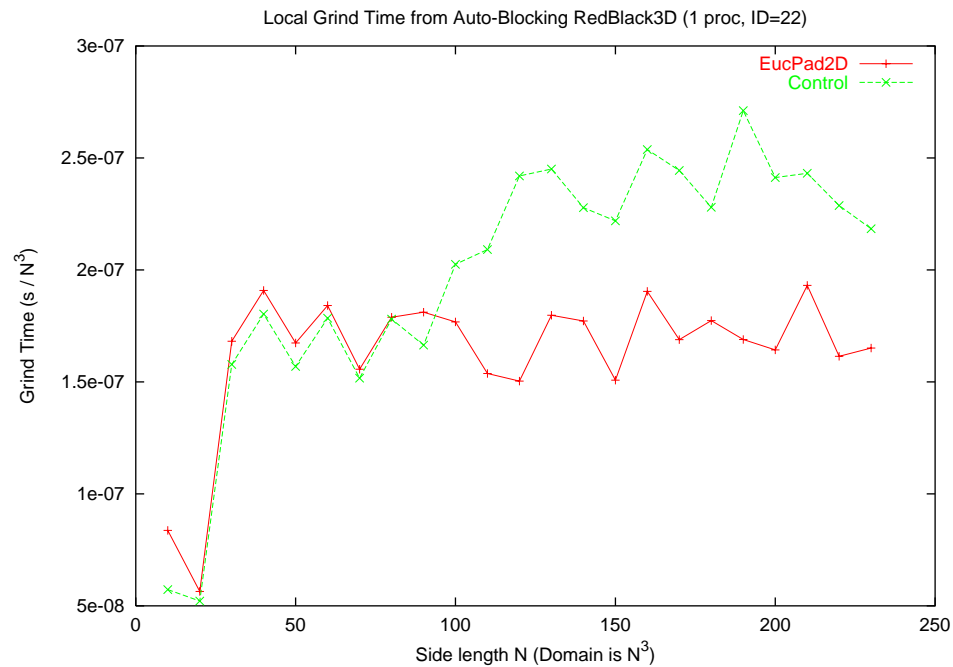
Figure 13. Grind Time from Tiling and Padding a 3D version of the RedBlack application on Beta. A simple modification to the EucPad2D tiling algorithm was used, which allowed three planes from the work array A to fit together in cache.

As expected, the performance varies significantly with the array size. However, we see a promising speedup for larger problems, up to 60% in some runs. The speedups are

made possible by a more TLB-friendly tile behavior in the 3D problems. Since the side lengths of the 3D array A are much lower, ranging from 10 to 230 instead of 1024 to 2048, the tile columns lie closer to each other in memory. As a result several columns can share the same TLB entry, and the negative effects we saw in the previous experiment are absent.

The tiled run follows the control for the first few problem sizes before pulling away around $N=100$. Since this algorithm does not reduce conflict misses by a significant degree, all speedups are due to a reduction in the number of capacity misses in cache.

The next set of graphs show the same experiment with the Autoblocker coordinating the tiling. As mentioned in the Design section, the principle difference between Fortran and Auto blocking is the simplicity of the auto-blocked numerical code. The Autoblock API is available for inspection in Appendix B. The first graph in figure 14 shows the Grind Time of the auto-blocked RedBlack3D, and the second graph shows the relationship between Fortran and Auto Blocked performance.



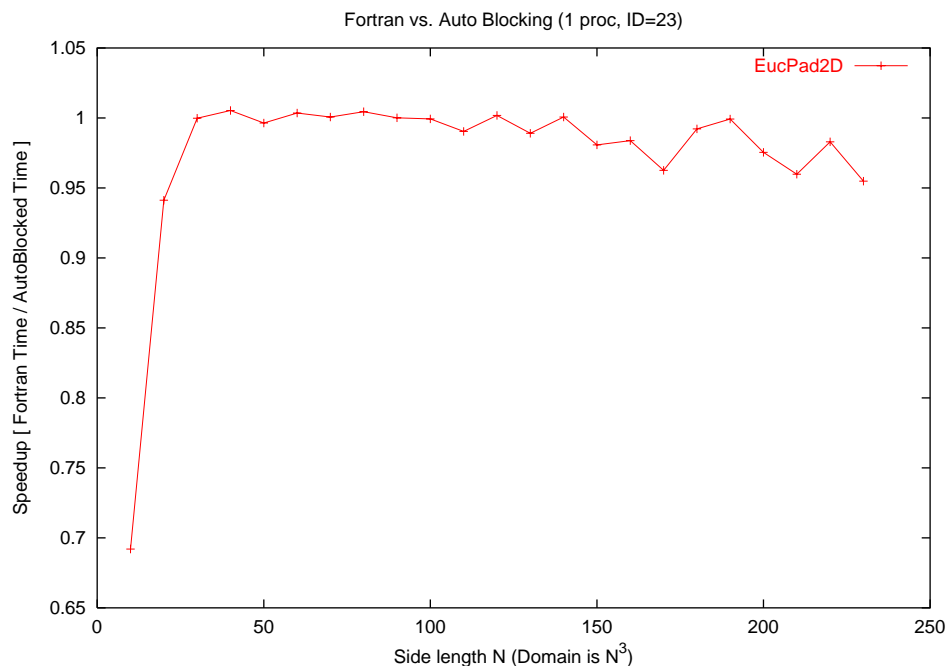


Figure 14. Auto-Blocked Redblack3D performance. The EucPad2D algorithm increases performance by reducing the number of capacity misses in cache, but does not prevent self-interference. The second graph compares Fortran Blocking with the simpler Auto Blocked approach, showing their similar performance.

The second graph makes it clear that Autoblocking does not slow the problem significantly. Since the autoblocker calls the RedBlack subroutine more times than the Fortran-blocked version, there is some extra overhead from setting up the fortran arrays and doing the jump-to-subroutine instruction once per tile rather than once per iteration¹³. However, as the recent experiment shows, this overhead is less than expected. We use the simpler autoblocked version of RedBlack for all subsequent experiments.

3. GcdPad3D Experiments

The next set of experiments use the GcdPad3D algorithm to choose tile sizes and pad the work array A. We expect to see fewer conflict misses, due to the aggressive padding

¹³ There are actually 2 calls to the kernel per iteration: one for the red points, and one for the black points in the array.

strategy of the algorithm. This experiment ran on Beta using the autoblocked RedBlack3D kernel.

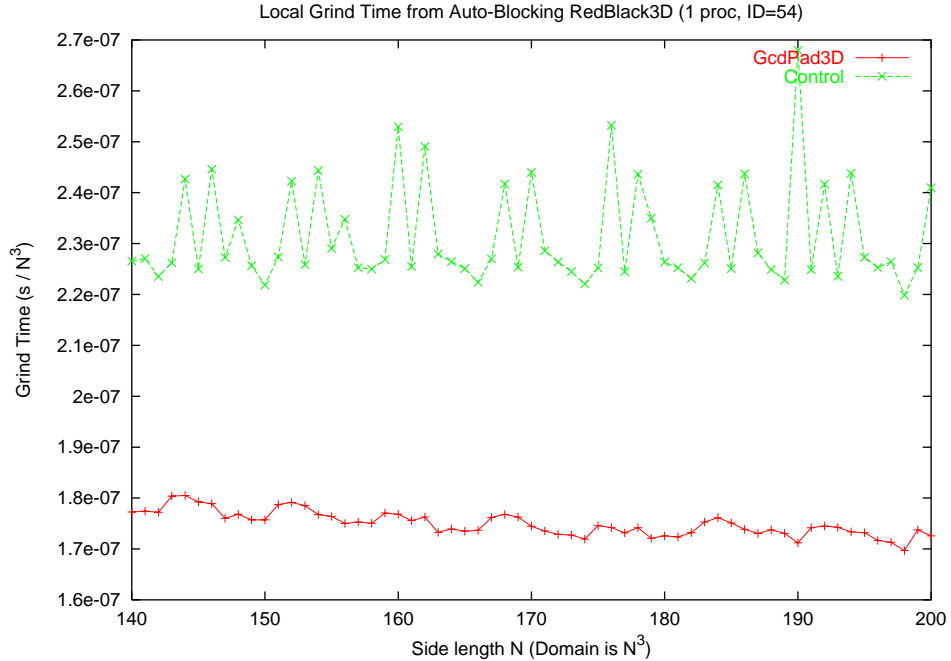


Figure 15. RedBlack3D Tiled vs. Control performance using GcdPad3D, on Beta. Note the smooth, linear performance-to-workload curve of the tiled run. The desirable performance comes at the cost of added memory overhead.

The Grind Time graph above shows the effectiveness of the GcdPad3D algorithm. Almost all of the performance/workload (P/W) irregularities present in the control experiment are smoothed by this tiling strategy. In the next chapter, we will use this linear P/W ratio to help load balance our parallel application. From the experimental results above, we conclude that an aggressive padding strategy is needed to minimize conflict misses in L1 data cache. Furthermore, since EucPad2D and GcdPad3D both use tile sizes that avoid *Capacity* misses in cache, but only GcdPad3D yields a linear P/W curve, we conclude that cache *Conflict* misses have a large impact on our application performance. This finding agrees with Lam, Rothberg, and Wolf's results in [17].

This next experiment was done on the Gamma cluster with 16 nodes. As expected, it yields similar performance to Beta's single processor results. The central difference between the machines lies in their processor architecture; Beta uses an AMD Athlon processor running at 850Mhz, while the Gamma nodes used have Pentium III 1Ghz CPUs. Since Gamma splits the problem between 16 processors, the problem sizes in the experiment are correspondingly larger; each node in the Gamma cluster receives approximately the same range of N values as for the experiment on the Beta machine.

The plot in figure 16 shows the effectiveness of the GcdPad algorithm when run on a parallel machine.

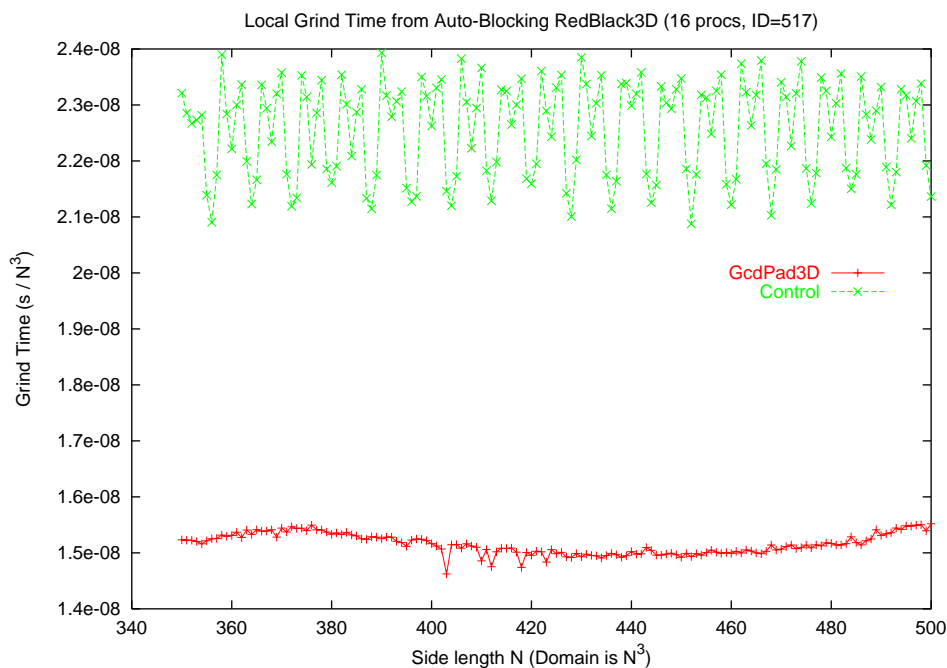


Figure 16. RedBlack3D Tiled vs. Control performance using GcdPad3D, on the Gamma cluster with 16 processors. The smooth predictability, and performance speedup from aggressively padding with GcdPad3D is evident here, as on the Beta machine.

The smooth performance we saw from GcdPad3D on Beta is evident on the Linux cluster as well. The difference in speedups is due to the different processor architectures

(AMD Athlon vs. Intel Pentium III), and the variations due to communication issues between the nodes of the cluster. We can see from the Grind Time plot that the Autoblocked GcdPad3D technique actually yields a slightly larger speedup on the cluster than on the single processor machine Beta. The important linear P/W relationship is still clearly present.

We now see a linear performance-to-workload curve from tiling our problem. GcdPad3D smoothes the performance irregularities due to cache effects, and we are ready to implement our load balancer on top of this tiling structure. In the next section, we summarize the results of this chapter, and discuss how it will integrate into the material in Chapter 3.

E. Discussion

In this chapter, we have explored our motivation for cache-friendliness, examined existing work in the field, and presented the theory, design, and experimental results from various cache tiling algorithms. Our reason for considering cache has always been to find reliable and predictable performance for a given amount of work. In the process, however, we have simplified the numerical processing code by tiling the problem and handling its communication ghost-cells with the Autoblocker, all done transparently with respect to the programmer. We have also realized a consistent 30-40% speedup over our non-tiled application by reducing capacity and conflict misses in the processor cache. In our final experiments, we found a tiling algorithm that met our goal of linear performance under varying workloads.

The factors that impeded the algorithms were due primarily to subtle effects in the cache and the processor's TLB. Conflict misses are difficult to predict, and can be hard to avoid. In the end, we resorted to an aggressive padding strategy that ensures tile columns and planes do not map to the same location in cache. The memory wasted through

padding would be a strongly discouraging factor, but is mediated somewhat by observations about virtual memory behavior.

The precise amount of physical memory overhead due to padding depends on the number of untouched pages in our virtual memory space. On our applications, arrays are arranged in column-major order. Since GcdPad3D pads significantly along the column axis (~ 200 extra elements or $\sim 2\text{KB}$ for experiments on Beta), much of the column padding will be mapped into physical memory. This occurs because while padding adds 2KB to the end of each column, the system page size is a larger 4KB . Therefore most padding elements will lie in a page containing some amount of real data. Because an entire VM page will be mapped to physical memory if a single address in it is accessed, most padding elements in a column will take up physical memory space, even though they are never used.

Although the padding elements along a column will increase our real memory footprint, the padding elements along a row (between planes) will not. These padding elements will reside in contiguous VM pages that contain no real data, and therefore will never be accessed by the application or mapped to physical memory by the OS. Therefore padding along a column is much more expensive than padding rows. We now calculate the actual memory overhead from padding with GcdPad3D.

We begin with a worst-case analysis. In GcdPad3D, the maximum padding along a column is $2T_1$, where T_1 is the length of a tile column. For a very large cache, or small problem size, T_1 could reach its largest possible value of A_1 . In this case, $A_1 = \text{gcd}(C, A_1)$. The algorithm will pad $2T_1$ or $2A_1$ elements along a column, and if we assume all these elements are mapped to physical memory, the padded problem will increase its memory footprint by a factor of three.

Simulations of GcdPad3D have shown that for medium to large problems, this overhead is small. However, 3D cubic problems smaller than 500^3 elements will see a

bigger penalty that can approach the theoretical worst case. Reducing the amount of padding overhead is a topic of future work.

The Pad algorithm [7] does reduce the padding overhead in many cases, but its increased time complexity make it inappropriate for our design. Rivera and Tseng's work used direct-mapped caches in their analysis and experiment. We are optimistic that by taking into account the higher cache associativity of modern processors, a similar algorithm can be developed that allows a certain amount of conflicts to occur in exchange for a lower padding overhead. The cache associativity hardware would absorb the extra conflicts without penalty, while the algorithm would insure we do not exceed the cache's conflict-avoidance limitations.

We have identified another factor that contributes to performance irregularities in our tiled experiments. Although the tiling algorithms insure a tile evenly divides the large padded array B, it is often the case that the tile does not do the same with the work array A. When this happens, we will encounter several smaller tiles than normal. We call these small ones *leftover tiles*, and although they do not cause more conflict or capacity misses than their normal sized counterparts, they have the same fixed subroutine-calling overhead but perform less work due to their smaller size. In practice, however, leftover tiles cause only small variations in performance, as evidenced by the smooth performance-to-workload curve in the GcdPad3D experiments.

In the next chapter, we integrate this cache-friendly work with a Hilbert Space-Filling curve-based load balancer [1]. Each node on the parallel cluster will receive multiple sub-problems, which we call *quanta*. The load balancer will move these quanta through the cluster as needed. For reasons presented next chapter, the quanta are themselves tiled by the autoblocker on each node. This two-level blocking strategy splits the problem into many fixed-sized quanta, and divides each quantum into multiple tiles (whose size varies with the local node characteristics). Throughout its design, our load balancer relies on the

predictable node performance established by the cache-friendly work presented in this chapter.

Chapter III

Load Balancing

A. Motivation

Distributing work effectively on a parallel computer is a difficult but important task. Programmers traditionally concentrate on writing applications correctly, and leave performance optimizations to the compiler and system hardware. Unfortunately, in parallel environments these automatic optimizations are not as effective. While in the single processor case a compiler can identify a computationally intensive *hot-spot* in a piece of code and optimize it to improve performance, it is much harder to do so among nodes of a parallel cluster. On a single processor, effective optimization relies on global knowledge of the code. On parallel systems, on the other hand, no node has global knowledge of the whole cluster. As a result, it is much harder to identify bottlenecks in the machine. Load balancing is a technique that detects these overloaded regions, and allows work to flow to other, less-pressurized parts of the system.

A fully dynamic load balancer frequently checks the total “pressure” of the system. Through its actions, this type of balancer allows a problem to flow like liquid through the machine, relieving bottlenecks and engaging unused resources. When all nodes in the cluster operate at their peak capacity (but not over it), we say the application is balanced. However, factors such as ad hoc work distribution, heterogeneous cluster hardware, and applications with progressively changing workloads make this a more complex problem than it appears.

Without careful thought, haphazard work distribution across the cluster can easily arise. It is well established that communication between nodes requires an order of

magnitude more time than communication within a single node. If data dependencies are not carefully measured and taken into account when partitioning the problem, nodes may communicate more data between themselves than necessary, harming performance. More often, these dependencies are difficult to determine *a priori*, and designers choose a naïve partitioning for simplicity.

We design load balancers to step in and take over this job of analysis and optimization, alleviating the burden on programmers. As data dependencies are often application-specific, many load balancers use an iterative strategy that monitors the application during runtime to record its behavior. Information gathered in this stage is used to make balancing decisions. We take this approach.

The structure of parallel clusters themselves adds another dimension to the load balancing problem. Clusters often grow in size over time, as budgets and needs allow. As component speeds increase, new processors are likely to be faster than the existing hardware they augment. This characteristic can cause applications to run inefficiently, even if they were partitioned with care. The slowdown arises from the structure common to many scientific applications. Frequent synchronization points built into the algorithms cause faster nodes to endure idle waits. Synchronization points force all nodes to arrive at a common point in the program before any node can continue. Variations in node speeds will cause fast nodes to arrive at a synchronization event first, and be forced to wait for their slower peers. Therefore load balancers must consider hardware heterogeneity.

Finally, changing workloads further complicate the load balancer's task. Even if it succeeds in balancing an application, algorithmic factors could cause workload changes, unbalancing the partition. Particle methods, and AMR-type problems have this property; hot-spots in the problem may move through the domain as the execution progresses. A load balancer must take these factors into account as well.

Much work has focused on how to solve these problems using *diffusion*-based distributed algorithms. With this approach, no omniscient master node is required. Each node uses only information from its closest neighbors to determine what work transfer needs to occur for a better balance. Our balancer, on the other hand, takes advantage of global performance knowledge to more effectively balance an application. We feel that by leveraging a small amount of global information, we can converge to an optimal balance more quickly than local-knowledge diffusion algorithms.

The rest of this chapter is organized as follows. In the next section we present existing work in the field of dynamic parallel load balancers. The design section discusses problem quantization, and explores the two-level blocking strategy used to make the cache-specific tile sizes compatible with the problem-specific quanta. The Hilbert curve we use to minimize inter-node communication is also presented. In the experiment section we describe our experiments and analyze the results. The Discussion section gives further analysis of the results and summarizes the chapter.

B. Existing Solutions

Much work has been devoted to load balancing in the field. Lin and Keller [32] studied diffusion techniques for load balancing, modeling their algorithm after pressures in a fluid. In his scheme, hot-spots in a parallel computation represent high pressure regions. The algorithm makes several passes of the cluster node graph, and on each iteration attempts to alleviate a measure of pressure by moving work. Cybenko was the first to apply rigorous linear algebra techniques to this diffusion approach, leading to his seminal paper [10] that proves an upper bound on the number of iterations required to converge on an optimum workload balance.

Cybenko's paper led to similar work by Diekmann, Frommer, and Monien [12] that employs linear analysis to improve the convergence properties of diffusion-based load

balancers. Diekmann later provides a summary [33], classifying balancing algorithms by the type of problem they suited for, including static, dynamic, and adaptive problems. Kumar et al. survey the scalability of load balancers [34], by computing upper bounds on their running time and number of messages sent. Their survey includes not only the mathematically interesting diffusion-based schemes, but also simple master/slave balancing approaches where a master node coordinates all balancing activities.

Flaherty et al [52] compare the performance of diffusion and re-partition techniques. Diffusion-based balancing, called *Migration* by Flaherty, follows Cybenko's approach where load is exchanged between neighboring nodes in the cluster. *Repartitioning* involves a global re-distribution of work. Both techniques are based on an Adaptive Mesh Refinement (AMR) Finite Element Analysis (FEM) application that refines its problem partitioning during computation. The Migration load balancer has a lower cost in terms of communication, but is less effective than full repartitioning since it uses only local knowledge of processor load. Repartitioning efficiently balances the application in one step, but requires global knowledge of processor load, and therefore has a higher overhead cost. The conclusion indicated that for the largest test, repartitioning was more effective than migration at reducing solution time. The authors identified the *interprocessor connection density*, or number of data dependencies between processors, as a key indicator of the Repartitioner's advantage over the Migration balancer. Our balancer follows the repartition design.

Space-Filling Curves [53] provide a mapping of partitions to processors such that problem locality is preserved. The DAGH partitioner [54] uses these curves as the basis of its design, as does the balancer by Pilkington and Baden [1]. The curve maps an N-dimensional problem onto a simple 1-dimensional curve, so that the interprocessor connection density is minimized. The mapping is also computationally easy to compute and invert.

Load balancing occurs by partitioning this curve among the nodes in a cluster. When applying a well-known geometric partitioning technique such as ORB [44] to a 3D problem, balancing inaccuracies may be introduced during the first iterations in each dimension. Using Space-Filling Curves, we are able to use these techniques to partition a problem more accurately due to its one-dimensional structure.

Certain flavors of Space-Filling curves, such as the Hilbert curve [1, 54], have strong locality properties. A Hilbert curve minimizes the *boundary face* surface area [52] of a set of partitions on a processor. This important characteristic reduces expensive inter-node communication within the cluster. A more detailed analysis of Hilbert curves and their application to load balancing appears in the next section.

Our balancer uses the cache-friendly techniques from Chapter 2 to insure maximal performance regardless of partition size. The parallel partitioner by Teresco and Flaherty [55] mentions cache performance as a potential benefit of their framework, which places multiple partitions on a processor in a similar fashion to our quantum design. They do not specifically use cache performance in their analysis, however, and make no attempt to ensure partitions are cache-friendly.

Graph partitioning involves an important class of problems defined by a graph of work elements. These problems are more general than the dense-array-based applications we analyzed in the previous chapter. Problems that have a non-uniform distance between elements fall into this category, and include modeling and simulation of physical objects as in Finite-Element-Analysis. Partitioning and balancing graph problems is a challenging topic that has received much focus in the literature [37, 38, 39, 41]. However as in the previous chapter, we restrict ourselves to dense-array problems. We note, however, many of the techniques used on graph problems such as Spectral bisection [42, 43] have been applied to dense-array applications.

Using Hilbert Space-Filling curves and our cache-friendly work from the previous chapter, we present experimental results from a using our load balancer on a parallel application. In the next section we describe the design of the liquid balancing algorithm, and its problem-quantization engine.

C. Design

Before presenting our balancer design, we describe our cluster model. The liquid balancer is designed for a parallel cluster with heterogeneous nodes that potentially have different cache sizes, processor architectures, and CPU clock speeds. Furthermore, we assume a non-uniform, dense-matrix problem that is partitioned using strictly rectangular shapes. Each rectangular partition covers a non-overlapping portion of the original problem, and run on a separate node in the cluster. Although partitions are often different sizes, together they conform to the partition problem:

$$\Lambda = \bigcup_{j=0}^P \lambda_j, \forall i, j | i \neq j \ (\lambda_i \cap \lambda_j = \emptyset) \quad (1)$$

Where Λ is the problem space, P is the number of partitions, and λ_i is a partition of Λ .

A partition in this model is relatively immutable. Once chosen, it may not change size without affecting the shape of all its immediate neighbors, a property that follows from (1). For several reasons, we choose to decompose the problem into finer-grained, rectangular partitions called *quanta*.

To balance a workload, we need to allow it to move through the machine. Since the problem may be non-uniform, certain elements may require more work to compute than others. If we moved work by changing the partition shapes, we effectively can move a small set of elements (a row or column of them), from one node to another. However, since elements can be non-uniform, their number does not tell how much work has

moved. A set of *heavy* elements will have a different effect on the balance than a set of *light* ones. Therefore, we make the assumption that the amount of time it takes to compute an element is unknown until we measure it. To ensure fairness, we base our measurement entirely on wall-clock timings taken from inside the application, as described later in the Autobalancer portion of this section. For any reasonably sized problem, obtaining timings of each individual element is not practical. Therefore, we time bundles of elements called quanta.

1. Problem Quantization

Quanta are small problem partitions. They conform to equation (1), and together completely define the problem space. Each node owns multiple quanta. The number of these per node is called the *Quantization factor* Q , and is determined by experiment. Since a quantum is individually timed, its elements have a known weight. To maintain the integrity of this value, we never change a quantum's size. After problem partitioning, the shape of all quanta are fixed for the duration of the execution.

A quantum is the unit of work movement. During a balancing operation, nodes may exchange a unit number of quanta between themselves, but never a portion of one. We can vary Q to improve load balance accuracy at the expense of increasing communication and other overheads, as we will see in later sections.

Careful readers will note that quanta act like cache tiles from the previous chapter. They effectively split a problem into many autonomous blocks, changing the locality distances of the problem. However quanta sizes are fixed, while tile sizes are based on the characteristics of each node. If we attempted to size quanta so they conform to tile sizes, we would quickly run into a paradox during balancing. A tile on node A may be smaller than on a neighboring node B. If A sends B a quantum during a balancing operation, B needs to increase its size for cache. However, since quanta are immutable, node B would have to live with a sub-optimal tile size.

To address this problem, we use a two-level blocking strategy. The immutable quanta represent the first layer, and cache tiles the second. Each quantum is tiled by the autoblocker, ensuring the tile is shaped for the specific local cache. This approach decouples the node-specific tile sizes from the problem-specific quanta.

2. Timings and the Autobalancer

In our model, a node has an unknown performance until it is timed. To accurately measure this value, we time individual quanta from *within the application* using a facility called the autobalancer. Taking measurements from within the program avoids problems with load-monitors such as the Network Weather Service [45], which use a separate process to measure performance. The monitoring process can be starved for CPU cycles or influenced in some manner by the target application, reducing the accuracy of its data.

The autobalancer is a special C++ iterator similar to the autoblocker described last chapter. In our KeLP programming model, all computation algorithms reside inside a standard iterator called *nodeIterator* that deliver one quanta at a time to the numerical kernel. The autobalancer is a *nodeIterator* that takes detailed timings for each quanta as they are computed in the iterator loop. In this way, we obtain quanta timings from inside the application without cooperation from the programmer.

After the autobalancer has collected a full set of local quanta timings on a node, it exchanges them with all its neighbors in the Timing Publish phase. This step requires P message broadcasts, where P is the number of nodes. These broadcasts publish the timings of *each quanta* to all other nodes. However, our experiments have shown that since the timing values are small (8 bytes for each quanta) the overhead from distributing the timing figures is much smaller than time needed to transfer actual data in the work-transfer balancing phase. Each node in the computation carries out the remaining balancing steps in parallel.

Once we have the freshly-acquired quanta timings we can make balancing decisions. In an unbalanced computation, we will notice that some quanta run faster than others. Specifically, the time/element value may be different among the quanta. To achieve balance, we strive to make the sum of quanta timings equal for all nodes:

$$t_i \in T \mid 0 \leq i \leq P, \quad t_i = \sum_{q \in Q_i} \frac{q}{|q|}, \quad \forall t_i, t_j \in T, \quad t_i = t_j. \quad (2)$$

Where Q_i is the set of quanta owned by processor i , and $q / |q|$ is the time to compute one element of a quanta in Q_i . P is the number of processors. The value t_i is the sum of quanta timings on processor i .

With this goal in mind, we present our Hilbert Space-Filling curve and RCB balancer design. The Hilbert curve specifies which quanta to move when balancing, and keeps inter-node communication to a minimum. The RCB algorithm makes a fair partition based on the quanta timings.

3. Space-Filling Curve

The Hilbert Space-Filling curve [1, 53, 54] is simply an ordering of the problem quanta. We use this ordering because Hilbert curves preserves problem locality. Quanta along the curve not only are adjacent in the problem domain, when assigned to nodes the boundary surface area between processors is minimized. We seek to reduce the surface area on processor boundaries because communication overhead depends on it. Each element on a node boundary will be exchanged with its neighbor on an adjacent node once per iteration (for the RedBlack kernel). Therefore the amount of node-node communication is proportional to the surface area of a node's partition. The Hilbert curve minimizes this surface area, and its self-similar structure enables it to maintain this property for an arbitrarily large problem. Figure 17 shows a Hillbert curve moving through a simple 2D quantized problem.

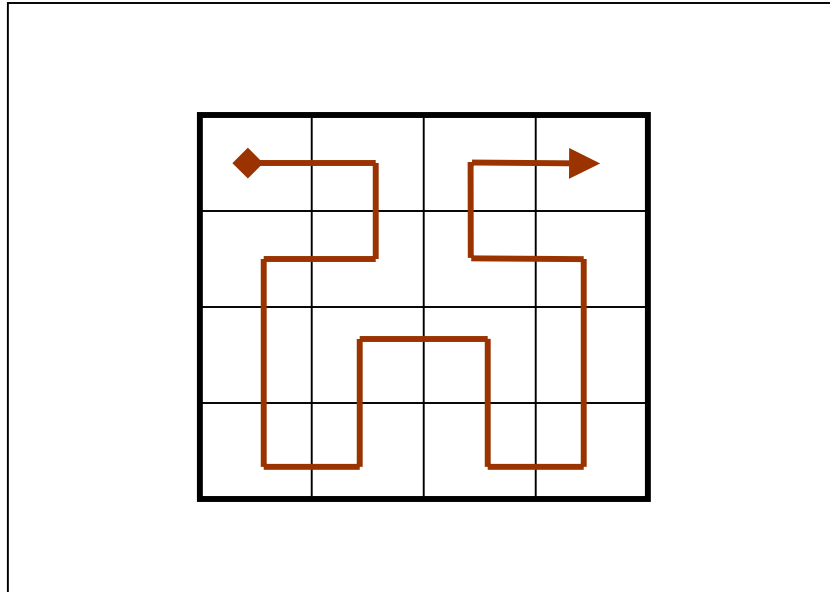


Figure 17. A Hilbert Space-Filling curve moving through a 4x4 array of elements. The ordering imposed by the curve tends to create square and cubic clusters of elements, a desirable property for reducing inter-node communication in the parallel cluster. Note that in larger arrays, the Hilbert curve makes larger versions of this shape, with this figure as every element. In this way a Hilbert curve as self-similar across scales, much like a fractal.

We use the RCB algorithm [44] to partition the Hilbert curve based on quanta timings. This well-known algorithm uses a simple divide-and-conquer method to create partitions of consecutive quanta such that we come as close to the balance equation (2) as possible. Since we never split a quantum, smaller quanta allow more accurate balancing due to its finer granularity. However, if a quantum becomes smaller than the optimum tile size for a node, cache efficiency may suffer. We discuss this balancing accuracy – tiling efficiency tradeoff in the next section.

Once the new partition has been calculated, we instruct the nodes to transfer quanta to other parts of the machine as necessary. Nodes can coordinate this task easily since each has global knowledge of the new partition. When a node A transfers a quantum to node B, it initiates a message to B containing all the elements of the quantum. When B

receives the message, it creates new local quanta and pads it according to its local cache characteristics. Node A then deletes its data, and the computation continues.

Our experiments have shown that this data-transfer phase takes much longer to complete than the balancing algorithm itself. Its overhead depends on the problem size and the degree of balance present in the system, and may represent a significant percentage of the overall running time. To justify the costs of load balancing, every transfer decision must have a beneficial effect on the computation. Therefore, we take a conservative approach to data-transfer. An inertial factor in our algorithm, called a *dampener*, favors smaller data transfers over large ones, in the hopes that we will avoid spurious transfers.

4. Dampener

Our balancer has a feedback-loop design. The output from a balancing iteration (which we call an *Epoch*) is fed back as the input for the next. This design naturally suggests itself for dynamic control problems such as ours. A consequence of the feedback-loop is that slight errors in the output can become magnified in successive epochs, leading to mistaken decisions. As we have pointed out, no balancing decision will be optimal because of the finite granularity of the quanta. Therefore, we need a mechanism to reduce the cost of balancing errors.

Many designs use a *weighting* strategy to deal with the sensitivity of feedback loops. New inputs are not taken as gospel, but instead contribute only a portion of their value relative to the previous input. In effect, this approach *dampens* the rate of change. Our design employs this idea. At every epoch, we receive from the balancer a statement of how quanta should move. This statement appears as a set of quanta-processor mappings. As in KeLP [3, 23], we call this mapping a *Floorplan*. To dampen the system, we compare each new Floorplan to the previous one as described in the following equation.

$$Floorplan_{i+1} = Floorplan_i + \alpha(Floorplan_i - Floorplan_b) \quad (3)$$

Where $0 \leq \alpha \leq 1$ is the dampening rate, $Floorplan_b$ is the proposed floorplan from the balancer, and $(Floorplan_i - Floorplan_b)$ is a set difference operation.

Effectively, this simple idea moves only a portion of the quanta suggested by the balancer. In the next section we demonstrate the importance of the dampening strategy when balancing real problems.

D. Experiment

This section presents experimental results from running our balancer on a parallel cluster. The first set of experiments determine the optimal Q factor, or number of quanta per region. The second set demonstrates the balancer operating at steady state. The remaining tests present the balancer in action, operating on a non-uniform 3D problem. We use the RedBlack3D application from last chapter for all our experiments.

The experimental setup is identical to last chapter. We choose the Gamma parallel cluster [27]. The cluster interconnect, the network connecting its nodes, plays an important part in these experiments. The Gamma cluster uses a Myrinet 2000 interconnect capable of 175MB/s throughput to connect its nodes. Experiments have shown its message startup time is around 7us. This network is optimized for MPI communication, which our RedBlack program employs.

On real computers, timings taken at the application level can occasionally be inaccurate. System interrupts from memory page faults, incoming network packets, and device requests can cause the inexact measurements. Since these system interrupts are transparent to the application, the program cannot account for them. To increase our timing accuracy, the balancer uses the median value of a small sample of 10 timings to make its decisions. Our expectation is that the median of this set will represent a clean

measurement, obtained during a period without system interrupts. Therefore, a full set of timings is made available to the balancer once every 10 iterations of the parallel application. The Autobalancer uses the *MPI_Wtime()* wall-clock function to measure timings, with an accuracy of around 1 μ s.

As in all our experiments, each plotted data point is the average of five independent executions, with outliers removed. Outliers are defined as any timing which is more than $\pm 2 * \text{Median}(5 \text{ Timings})$. In practice, we rarely have to discard outliers, and the figure used is the average of five runs. We measure performance as before in terms of Grind Time, the computation time per element.

1. Q-Type Experiments

These experiments determine the optimum value of the number of quanta per processor, or Q , from a performance perspective. In the previous section, we mentioned the tradeoff between large quanta that can be tiled effectively by the Autoblocker, and small quanta that will yield more accurate balancing. The experiments presented here give evidence for the tradeoff, and show that communication time plays a large role in the choice of Q . These results consider only the performance characteristics of the application for different values of Q . We use these results to pick a Q value for our balancer, and later analyze the balancing accuracy that this choice affords.

By the results of last chapter, we expect to see a performance speedup from tiling and padding the application. Although increasing Q will permit better load balance, we anticipate an additional running time overhead as well. In the interest of maintaining a performance speedup, we choose a value of Q with a run time within 10% of the best case.

In this experiment we pick a single RedBlack3D problem size, 320^3 , and measure its performance as we increase Q . The first y-value in figure 18 ($Q=0$) is the control experiment, against which we compare our running time. The remaining points

correspond to the performance of the quantized RedBlack3D application. Each quanta is tiled and padded using the GcdPad3D Autoblocker. The experiment is conducted on 8 processors of the Gamma cluster.

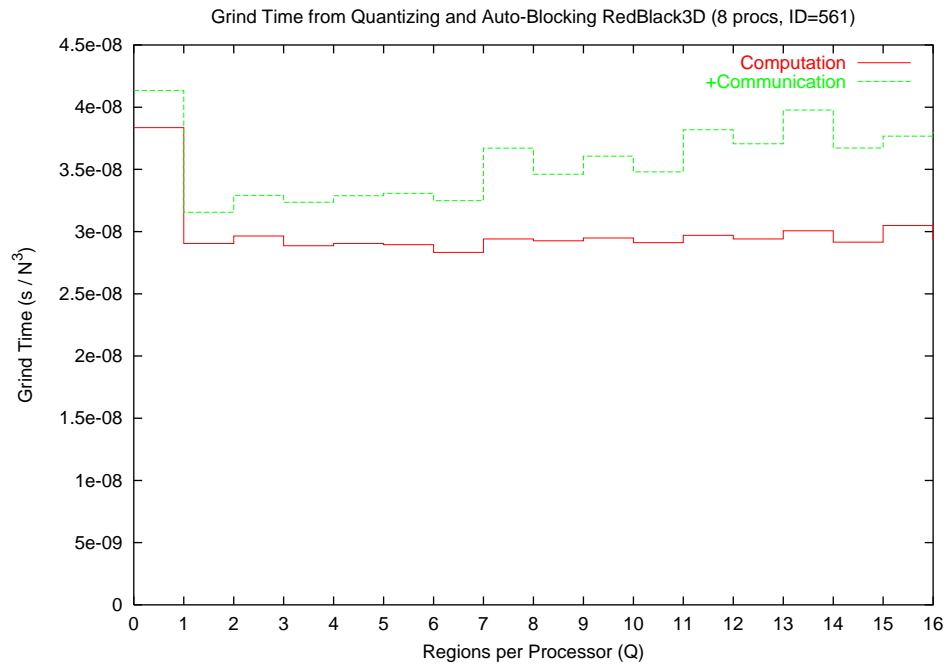


Figure 18. Grind Time vs. Q experiment run on 8 Processors of the Gamma Cluster. The *Computation* curve represents the time required to compute a point in the problem domain; the *+Communication* series adds the communication time for each point. The first point at $Q=0$ represents the non-tiled control run, which is provided as a reference.

Although the quantization factor (Q) has little effect on the local computation time, it does have a large impact on the communication overhead of the problem. This experiment shows that communication requirements limit the number of quanta we will assign per processor.

As the local computation time remains relatively constant, we conclude that the tiling techniques can handle small quanta sizes effectively. Although cache efficiency is slightly lower for large Q , because the small quanta prevent tiles from achieving their

optimal size, the results point to communication overhead as a more significant performance factor.

As the number of quanta per node increases, so does the time needed to update the communication boundary cells (ghost-cells) between quanta. We see from the Communication curve that this overhead significantly impacts performance. Using our requirement that performance remain within 10% of the best case, we determine an upper threshold for Q that qualitatively allows enough quanta balance effectively, while keeping communication overhead within acceptable limits.

As long as Q remains less than 8-10, the time needed to fill the communication cells surrounding each quantum remains acceptably small, and the running time is within 10% of the ideal ($Q=1$) performance. When Q increases beyond this threshold, however, we consider the communication overhead prohibitive. From a performance perspective, we would like to choose a small value of Q . However, since our ability to balance effectively increases with more quanta, we choose the parameter $Q=8$ for the balancer experiments. Although we chose this figure somewhat arbitrarily from these results, we present evidence that this parameter choice provides enough granularity to accurately load balance our application.

2. Uniform Balancer Experiments

To establish a baseline, we show the balancer operating on a uniform problem. This RedBlack3D application is already in balance, and we show that the balancer detects this state, and takes no further action.

To measure the effectiveness of the balancer, we introduce the *Balance Efficiency (BE)* metric [1]. The Balance Efficiency ranges from 0-100% and is highest when all processors compute an iteration of the parallel algorithm in identical time.

$$BE = \frac{\sum T_i}{P * \text{Max}(T_i)}, \quad 0 \leq i \leq P \quad (4)$$

Where T_i is the sum of all quanta timings on a processor i , and P is the number of processors.

The graph below shows the balancer operating on a uniform RedBlack3D problem. It presents Balance Efficiency vs. Epoch for an 8-processor run with 8 quanta per processor. The balancer Epoch represents one balance operation performed once every 10 iterations of the application. This uniform problem has 64 equal-sized quanta and all quanta require the same computation time.

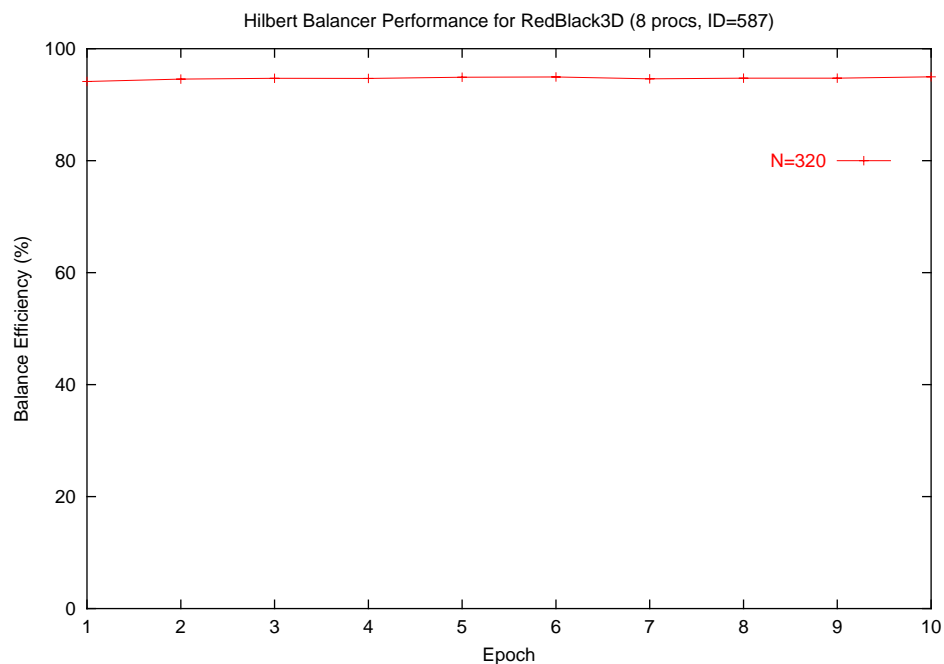


Figure 19. The balancer at rest. When we run the balancer on a balanced problem such as this uniform 320^3 RedBlack3D application. Each of the 8 processors own 8 problem quanta. The balance efficiency does not change during the experiment, illustrating the stability of the balancer. The dampener was set at 0.5 for this experiment.

The Balance Efficiency remains constant at 94% throughout the execution. The importance of Figure 19 is the stability of the balancer during the duration of the run.

Although dependent on clean wall-clock timings to operate correctly, the balancer does an adequate job of gauging processor load. We do not achieve perfect efficiency due to variations in the timings gathered by the balancer. We note that printing the *timing figures themselves* causes some variation because only the first processor generates the output. Therefore the first processor runs slightly slower than its peers, and causes a slight timing variation.

During this run, the Gamma cluster was under heavy use. When we ran the experiment with no dampening, the balancer chose to move quanta, though there was no need to. We concluded the heavy load caused spurious timing variations in the cluster. In response, we increased the dampener value to 0.5, which successfully brought the balancer to rest.

In the next section we present the balancer in action, actively moving problem quanta between processors during runtime. The next experiments show how the balancer brings an unbalanced application into balance, leading to a reduction in running time.

3. Non-Uniform Balancer Experiments

In this set of experiments we run the balancer on an unbalanced problem to observe its operation. The problem we choose is a non-uniform version of RedBlack3D. In this problem, only some of the quanta have real work, while the others have only light computational requirements (see figures below). Although this toy problem is easy to balance statically upon inspection, we note that our balancer detects the imbalance at runtime without any aid from the application itself. Therefore, we expect this experiment to show that our balancer can indeed detect and correct load imbalances automatically.

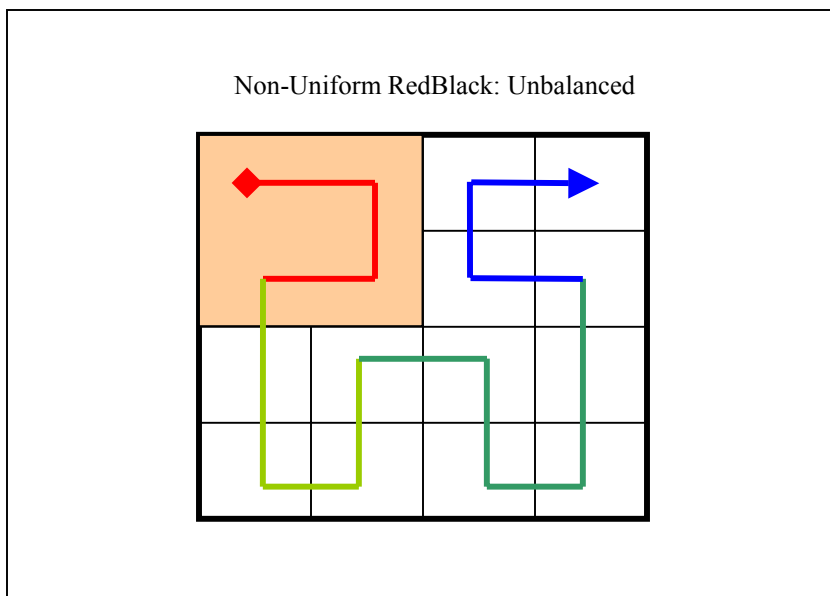


Figure 20. Processor assignments in a 4-processor non-uniform RedBlack problem before balancing. Each color along the Hilbert curve represents one processor. The shaded portion of the problem requires more work than the un-shaded regions.

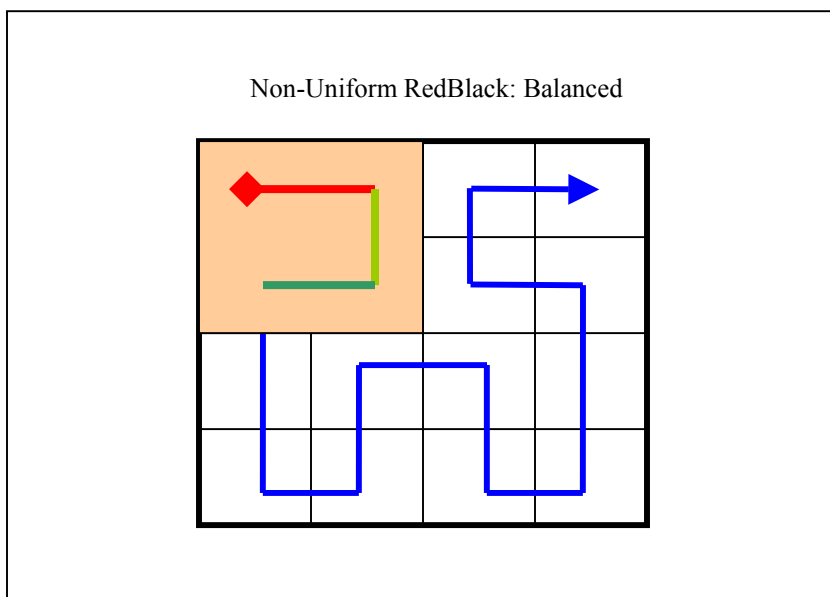


Figure 21. The non-uniform RedBlack problem after balancing. The shaded busy section of the problem receives more processors than the empty area, leading to a better balance. An actual balanced partitioning is presented in Appendix A.

The plot below shows the balancer in action; aggregating all quanta in the unloaded region of the problem onto one processor, and splitting the loaded portion of the computation between the remaining processors. We again chart Balance Efficiency vs. Epoch.

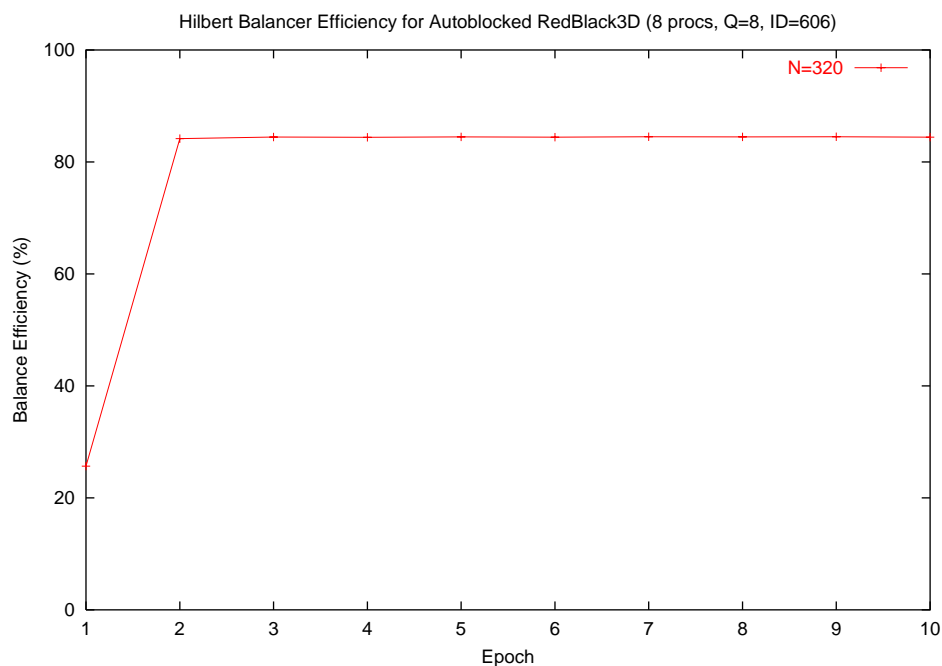


Figure 22. The load balancer in action. The non-uniform problem is quickly balanced, and the balancer rests for the remainder of the execution.

The balancer works quickly to achieve balance, making all changes in the first epoch. Again, the later epochs in figure 22 show the stability of the balancer. Although it does improve the Balance Efficiency significantly, the final achieved balance is less than we saw in the previous experiment. Despite the imbalance, however, the balancer detects it can do no better, and makes no further attempt to transfer work.

As mentioned earlier in the chapter, accurate balancing requires fine-grained quanta, as a single quantum is the smallest amount of work we may move during balancing. The tradeoff between numerous small quanta for balancing and fewer (but large) quanta for

communication led us to choose $Q=8$, as discussed in the previous section. The balancer achieves a BE of 84% in this experiment, which we believe is close enough to the 94% in the uniform case to demonstrate the effectiveness of our design. In the next section, we demonstrate the performance benefits from balancing this problem.

The load on the cluster was light during this experiment, and no dampening was needed. The following graph shows the speedup due to our balancing efforts.

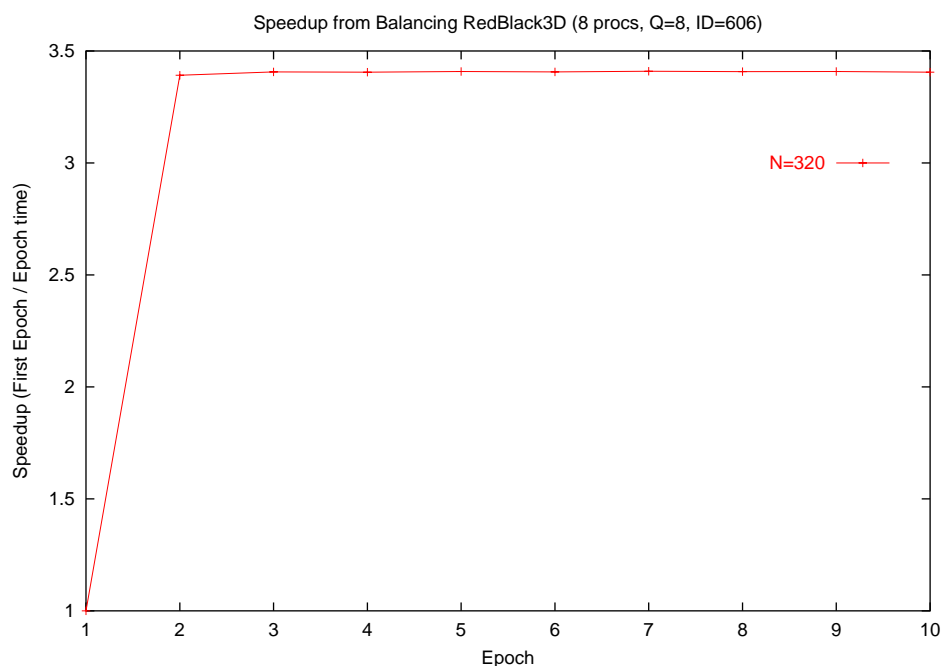


Figure 23. Speedup from balancing. We calculate the speedup as (*First Epoch Iteration Time / Epoch Iteration Time*). Timings taken every 10 iterations of the 320^3 -sized RedBlack3D problem, which was run on 8 processors of the Gamma cluster.

The speedup from balancing is large, as we would expect. Before balancing, the cluster required 0.50 seconds to complete 10 iterations of the RedBlack3D application. After the first balancing epoch, this figure becomes 0.15 seconds, a speedup of 335%.

The balancer requires computational and communication overhead to function, which we discuss in the next section.

4. Balancer Overhead

The balancer imposes an overhead to the running time of the application due to several phases of its operation. The most costly of these is data-transfer, where all data in a set of quanta are moved from one cluster node to another. The balancing algorithm itself requires a measure of computation and communication, and is another source of overhead. We separate the balancing algorithm overhead into two parts. First, the time required to distribute the quanta timings to all processors. We call this the *Timing Publish* phase. The second phase, where all nodes compute the RCB algorithm on the timings, we call the *Balancing Algorithm* phase. The two phases of the algorithm decide what quanta to move, and where to move them. The data-transfer phase invokes the MPI message calls to actually transfer the data between nodes.

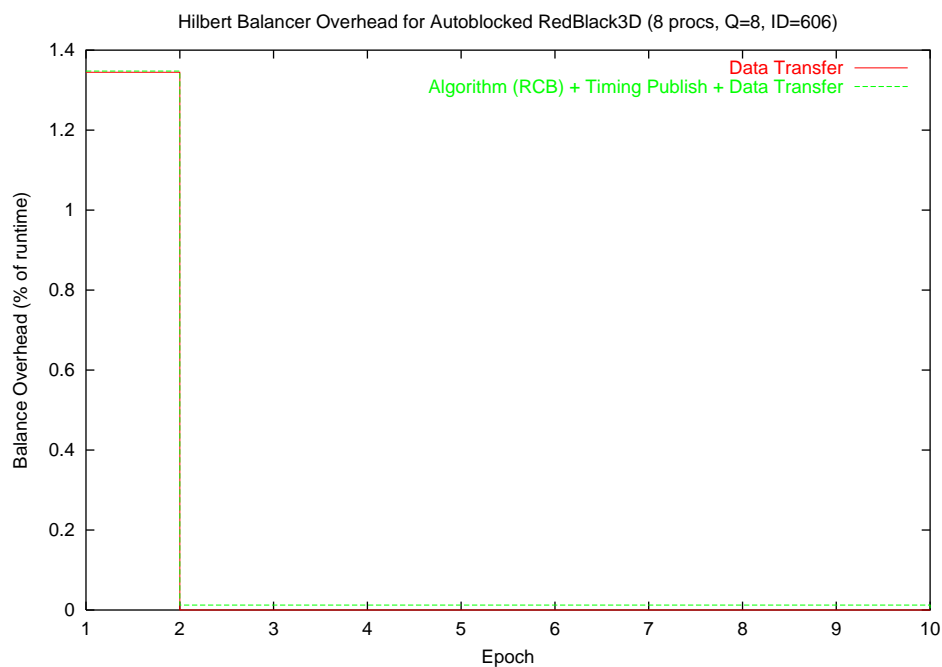


Figure 24. Balancer overhead. All values are given as a percentage of total run time. The red line shows the time needed for Data Transfer only, while the green set adds the Timing Publish and RCB algorithm overheads.

The balancer overhead is small compared to the overall running time of the application. As we expected, the overhead from the Data Transfer phase is much higher than either the Timing Publish or the Balancing Algorithm. This is in part due to highly optimized Timing Broadcast and RCB implementations used by the balancer. We see that after the data transfer in the first epoch, the overhead from later balancer epochs is negligible. During data transfer, we only measure the time needed for the MPI messages to complete. Other overheads such as memory allocation on nodes receiving data, and memory free operations on nodes sending data do not contribute significant overhead compared to message passing.

5. Irregular Balance Experiments

The experiments presented in this chapter have load balanced parallel problems with regular decomposition. The partitions of a regularly partitioned problem have identical size and shape, and each covers the same number of elements. By extension, all quanta in the computation have the same size and shape as well. As mentioned earlier in this dissertation, an important class of problems are *irregularly partitioned* so their quanta have varying shapes. An interesting experiment for future work will show how these problems have a natural load imbalance due to variances in the cache miss rates during computation over quanta of different shapes.

Irregularly partitioned problems allocate an equal measure of elements to each partition. We argue that because partitions have differing shapes, the cache miss rate will vary between them, leading to a load imbalance. By figure 1 we see that performance varies with partition size. The EucPad3D and GcdPad3D experiments in Chapter 2 establish that cache conflict misses cause this observed variation. We know the conflict miss rate is sensitive to memory access strides imposed by partition shape, since we achieved stable performance only when we pad as a function of the partition dimensions.

Without the application of cache-friendly techniques, the partitions of an irregularly-decomposed problem will experience differing cache miss rates. In figures 14 and 15 from last chapter, we see that the rate of cache misses significantly affects computation time. Therefore, an irregularly-decomposed problem will naturally have a load imbalance because the performance of its partitions will be non-uniform. By the results from last chapter, we believe our cache-friendly parallel framework will bring such problems into balance by minimizing cache conflict misses over all partitions, independent of their size or shape.

In this section we described our experimental setup and presented our results. In the next section, we discuss our findings and summarize the chapter.

E. Discussion

In this chapter we presented the motivation, design, and experimental results from our dynamic liquid load balancer. We leveraged our cache-friendly work from the previous chapter to obtain good performance for all quanta sizes.

The plentiful work on load balancers in the literature allowed us to choose the balancing technique most suited to our environment. The Hillbert Space-Filling curve guides our work movement during balancing, maintaining problem locality at all times to minimize expensive inter-node communication. The popular RCB algorithm partitions the 1D curve among nodes based on the application-measured performance timings.

We divide our problem into fixed-sized Quanta, which are the unit of performance monitoring and work movement in the balancer. We tile each quantum individually using cache-friendly techniques to ensure reliable performance, independent of quanta size. Our Autobalancer obtains reliable per-quantum timings of the Fortran numerical kernel without aid from the programmer. When all quanta are timed, and outliers removed, nodes exchange timing figures so each has knowledge of its peer's performance. Each node runs the RCB balancing algorithm in parallel. We keep a history of the *pivots*, or cut-points picked by the algorithm in each epoch. This cut-history is used to dampen the balancer decision to reduce the effects of balancing errors.

The dampener helps desensitize the balancer to load variations on the cluster from other jobs. This type of contentious load can lead to balancing errors that cause expensive data transfers to occur. To combat this effect, we use the dampener to keep the balance delta small, as described in equation (3). Exploring the precise effect of other jobs in the cluster on our balancer remains as an interesting area for future work.

Instance: List of Quanta. Floorplan[1:QP]: Floorplan[i] = (p, O, q_{start}, q_{end}), p is partition coordinate, O is processor owner: $1 \leq O \leq P$, where P is the number of processors and Q is the number of quanta per processor. The $q_{\{start, end\}}$ fields are quantum bounding coordinates. Coordinates are N-dimensional points in Euclidean space. **List of Timings.** T[1:QP]: T[i] = Quanta timing in seconds. **List of Cut Points** from previous partitioning. Cuts[1:P]: Cuts[i] = Index of processor ownership boundary in the floorplan.

Pre-condition on Input: TimingPublish(T[1:Q])

Balance(T[1:QP], Floorplan[1:QP], cuts[1:P]):

```
scan[0] = 0; scan[i] = Sum(1..i, T[i]), 1 ≤ i ≤ |T|
Floorplan' = RCB(Floorplan, scan, cuts)
MoveQuanta(Floorplan', Floorplan)
```

RCB(F[a:b], scan[0:QP], cuts[1:P], i=0, j=0):

```
if  $2^i \geq P$ :
    F[a:b].owner = j
    return
key = (scan[b] - scan[a-1]) / 2 + scan[a-1]
cut = find index of key in scan[a:b]
cut = Damp(cuts[j], cut)
rcb(F[a:cut], scan, cuts, i+1, 2j)
rcb(F[cut+1:b], scan, cuts, i+1, 2j+1)
```

Damp(old, new):

```
return old +  $\alpha$ (new - old)  $0 \leq \alpha \leq 1$ 
```

MoveQuanta(F' [1:QP], F[1:QP]):

```
for i in (1..QP):
    src = F[i].owner
    dst = F'[i].owner
    if src ≠ dst:
        transfer data in quanta i from src to dst
```

TimingPublish(T[1:Q]):

```
AllTimes = [1:QP]
for root in P:
    AllTimes ← broadcast(T, root)
return AllTimes
```

Figure 25. Balancer Pseudocode. Processors always own contiguous subsets of the floorplan. Notice the RCB scan array is calculated only once. The order of quanta in the floorplan follows a Hilbert Space-Filling curve.

We have established that the overhead from the Timing Publish phase is negligible compared to the cost of transferring quanta and their data payload during the work-transfer phase. Therefore we feel the quick convergence of our balancing algorithm (one iteration in our experiment) justifies the added cost of publishing the timing figures to every node, which requires $O(P)$ message broadcasts as described in figure 25. We note that our experiments involved only a small number of processors. If the scale of the cluster approached hundreds of nodes, the Timing Publish overhead may become prohibitively expensive, and a more distributed algorithm would be needed. Diffusion algorithms [10] fall into this category, and trade lower timing communication overhead for slower convergence.

Our balancing algorithm was chosen for its quick convergence. We chose this design parameter with the expectation that data transfer would be the bottleneck during balancing. Although our experiments confirm this, the Myrinet network in the Gamma cluster operates more efficiently than we had expected. In our experiment, it completed a data-transfer operation involving the communication of 54 quanta, each with 4MB of data, in approximately 2 seconds. The corresponding transfer rate is 110MB/s, which compares to the maximum measured Myrinet throughput of approximately 175MB/s. Standard 100Mbps Ethernet can achieve a theoretical maximum of 12.5 MB/s.

The observed transfer rate during balancing represents the actual message transit time, and two sets of memory copies, one each on the sending and receiving node. As the memory copies involve 3D data structures, we can expect many TLB and cache misses during the operation. Therefore, the 110MB/s effective data-transfer rate is quite impressive, and suggests that using a diffusion-based algorithm would be practical if the Timing Publish overhead becomes excessive.

The experiments illustrate the tradeoff between large and small quanta sizes. Since we balance by moving whole quanta, smaller quanta size leads to more accurate

balancing. However, if a quantum becomes smaller than the ideal cache tile size (a function of the $\sqrt{\text{cache size}}$), cache efficiency suffers somewhat. More importantly, however, numerous small quanta lead to prohibitive communication overhead. Each quantum has a layer of communication elements surrounding it, which we must update in each iteration. Therefore smaller (and more numerous) quanta require greater communication overhead per iteration. This effect is particularly evident in the Q-type experiments, which show communication time increasing with the number of quanta per processor. The quanta-size tradeoff led us to choose a Q value of 8, which we feel strikes a reasonable balance between communication overhead, tiling efficiency, and balance accuracy.

In the next chapter we conclude the thesis. We summarize our work and offer directions for future research.

Chapter IV

Conclusion

This thesis presents our cache-friendly liquid load balancer. We chose among several cache tiling and padding techniques for use in a two-level work granularity framework that achieves consistent parallel performance independent of problem partitioning. Although computation performance varies with partition shape, by applying our cache-friendly strategy to each partition we achieve maximum performance irrespective of the partition dimensions. We say our balancer is liquid because its dynamic and iterative monitoring allows work to flow fluidly through the machine during computation.

The Cache-Friendly chapter analyzed a variety of cache tiling and padding techniques. We showed by experiment that only aggressive padding between the columns and planes of a 3D work array was sufficient to reduce cache conflict misses in the L1 data cache. Although this padding leads to a memory overhead, we show that UNIX Virtual Memory behavior mediates the actual padding cost in terms of physical RAM.

When cache conflict and capacity misses were reduced, our experiments showed stable performance under varying partition sizes. Moreover, the runtime tiling and padding approach we chose led to at least a 30% speedup on every machine we tested. We point out that our tiling library can adapt to various hardware configurations more easily than compile-time techniques, since it can access vital cache statistics at runtime. Since we calculate the cache tile size and padding factor at runtime, our library may be designed to automatically adapt to the system hardware. Although in its current form, our framework does not attempt to automatically identify the local cache characteristics, its

runtime structure makes such a design feasible. Compiler-based tiling, on the other hand, minimally requires a recompilation for each new hardware architecture. For heterogeneous parallel clusters that potentially contain nodes with different cache sizes, a runtime tiling strategy with an automatic cache discovery mechanism seems necessary. Our work provides the first step towards such a design.

The Liquid Load Balancer builds on the reliable performance yielded by our cache-friendly efforts. When partition performance is independent of problem size, the first-level Quantum Partitioner can choose partition sizes freely, without concerns that pathologically shaped quanta will lead to poor performance. We identify a tradeoff between small quanta that aid balancing efficiency, and large quanta that lead to more effective cache tiling and lower communication costs.

We use a Hilbert Space-Filling curve to guide data-motion through the cluster so that inter-node communication dependencies are minimized. Minimizing communication is important due to the high cost of message passing. We handle non-uniform workloads by measuring node performance over fixed-sized problem quanta. Balancing occurs by moving these quanta between nodes in the machine.

In conclusion, our load balancer can successfully balance a non-uniform parallel application on a Commodity Cluster Computer, with performance insured over a range of parameter choices by our cache-friendly framework.

A. Future Work

This thesis gives rise to several future research possibilities. The tiling and padding algorithms impose a memory overhead, which we have not proved is minimal. Whether we may reduce this memory cost while maintaining the reduced capacity miss behavior is an open research question. We postulate that high cache associativity present in the

processors of our Gamma cluster may allow padding algorithms to trade a controlled rate of cache conflicts for reduced memory overhead.

For hardware designers, we ask whether the conflict misses shown in our results can be avoided altogether in hardware. We have presented a motivating example for which current commodity cache designs falter, leading us to use memory-wasteful strategies such as padding to achieve optimum performance.

Our balancing algorithm works well for small numbers of processors. However, it remains a topic of future work whether our balancer design is appropriate for larger parallel jobs. Due to our optimized Timing Publish algorithm and our balancer overhead results, we expect our design to scale well to tens of processors. However experiments with hundreds of processors may strain the scalability of our design.

We have only tested the balancer with a static non-uniform problem. The design can theoretically handle dynamic problems, and the quick convergence of our algorithm supports this belief. Applying our balancer to adaptive mesh refinement, particle methods, shockwave simulations, and other dynamic problems is an area of future work.

Finally, our balancer incorporates a dampening factor. This design feature is intended to allow the balancer to operate in a varying environment where load influences from applications can effect the accuracy of the node performance timings. Our analysis of the dampener is primarily theoretical, and we did not conduct rigorous experiments with respect to load variations caused by other jobs in the cluster. Such work remains as an area for future research.

I would first like to thank Scott Baden, my advisor, for his help, direction, and insight throughout this research. Greg Balls, Dan Shalit, and Bill Kerney in the Scientific Computing Group of the High Performance Computing Laboratory at UCSD have given numerous suggestions and useful analysis that helped shape this work. Eric Tune

provided invaluable hardware and architecture knowledge. Michelle Mills Strout, who has worked with caches far longer than I have, kindly offered her help for tiling algorithm analysis. Phil Papadopoulos, Mason Katz, and Greg Bruno of the NPACI Rocks cluster team were a great help and provided me with time on the high-performance Meteor cluster at SDSC. Finally I would like to thank my mother Wendy for pushing me to embark on this journey through the CS Masters Program at UCSD, which has proved a wonderful and cherished experience.

Appendix A: Tabular Data

Tile Size

The following tables list the tile dimensions chosen by the EucPad3D and GcdPad3D algorithms. Tiles are sized for a cubic array with side length N , and a 256KB cache size.

EucPad3D

Below is shown the tile sizes chosen by the EucPad3D cache tiling algorithm. The tile size is a function of the cache size and array dimensions. A cost function chooses the best tile from a set generated by adding up to 8 padding elements to the column length of A . We only present data for problem sizes greater than 100^3 . The *effective N* value represents the column size of A after padding and adding the KeLP ghost cells.

N	Tile (T_1, T_2)	Column Padding (effective N)
100	100,76	0 (102)
110	110,69	0 (112)
120	121,63	1 (123)
130	131,58	1 (133)
140	62,109	2 (144)
150	151,50	1 (153)
160	78,95	3 (165)
170	82,91	0 (172)
180	86,85	2 (184)
190	85,81	1 (193)
200	94,75	6 (208)
210	101,71	7 (219)
220	98,69	3 (225)
230	78,98	8 (240)
240	73,96	3 (245)

GcdPad3D

The following table lists the tile sizes and padding chosen by the GcdPad3D algorithm. The tile size is a function of the cache size, and the padding depends on the array size. We present the data points from figure 15, in the range 140 to 200. The (B_i, B_j) padded dimensions represent the column and row lengths of A after padding and adding the KeLP ghost-cells.

N	Tile (T_i, T_j)	Padded dims (B_i, B_j)
140-190	126,62	384,192
191-200	126,62	384,320

Q-Type Experiment

Below is shown tabular data from the Q-type experiment on the Gamma Cluster with eight processors (ID=561). Each value is the average of five independent runs.

Q	Computation (s)	Communication (s)	Total (s)
1	19.037	1.642	20.679
2	19.432	2.138	22.422
3	18.916	2.289	21.206
4	19.034	2.514	21.549
5	18.974	2.700	21.675
6	18.567	2.725	21.293
7	19.274	4.781	24.055
8	19.174	3.504	22.679
9	19.325	4.311	23.636
10	19.076	3.732	22.809
11	19.467	5.557	25.025
12	19.274	5.011	24.286
13	19.704	6.361	26.065
14	19.108	4.956	24.065
15	19.980	4.711	24.692
16	19.189	5.722	24.911

Balancer

The results from the balancer at rest experiment (ID=587) in tabular form. Each value is the average of five independent runs.

Epoch	Balance Efficiency (%)	Total Overhead (s)
1-10	94.500	0.000

The results from the balancer in action experiment (ID=606) in tabular form. Again, each value is the average of five independent runs.

Epoch	Balance Efficiency (%)	Total Overhead (s)
1	25.673	2.093
2-10	84.500	0.019

FloorPlans: Unbalanced and Balanced

The two KeLP floorplans shown below describe the 3D RedBlack quantum partitionings before and after balancing. The floorplans describe the quanta-to-processor mapping of the parallel problem. The application has a non-uniform structure as illustrated in Figure 21. Both floorplans cover a 320^3 -element cubic problem. The quanta order follows a Hilbert Space-Filling curve through the problem partitioning.

Before balancing, the unbalanced partition shows each processor owns an equal number of quanta:

KeLP FloorPlan: Processors: 8, Q: 8, Size: 64, Shape: (4,4,4)
id:(3D partition coordinate) processor owner:[quanta region extents] (size)

0:(1,1,1). 0:[(1,1,1),(80,80,80)] (512000)	32:(3,4,2). 4:[(161,241,81),(240,320,160)] (512000)
1:(1,2,1). 0:[(1,81,1),(80,160,80)] (512000)	33:(3,4,1). 4:[(161,241,1),(240,320,80)] (512000)
2:(2,2,1). 0:[(81,81,1),(160,160,80)] (512000)	34:(3,3,1). 4:[(161,161,1),(240,240,80)] (512000)
3:(2,1,1). 0:[(81,1,1),(160,80,80)] (512000)	35:(3,3,2). 4:[(161,161,81),(240,240,160)] (512000)
4:(2,1,2). 0:[(81,1,81),(160,80,160)] (512000)	36:(4,3,2). 4:[(241,161,81),(320,240,160)] (512000)
5:(2,2,2). 0:[(81,81,81),(160,160,160)] (512000)	37:(4,3,1). 4:[(241,161,1),(320,240,80)] (512000)
6:(1,2,2). 0:[(1,81,81),(80,160,160)] (512000)	38:(4,4,1). 4:[(241,241,1),(320,320,80)] (512000)
7:(1,1,2). 0:[(1,1,81),(80,80,160)] (512000)	39:(4,4,2). 4:[(241,241,81),(320,320,160)] (512000)
8:(1,1,3). 1:[(1,1,161),(80,80,240)] (512000)	40:(4,4,3). 5:[(241,241,161),(320,320,240)] (512000)
9:(2,1,3). 1:[(81,1,161),(160,80,240)] (512000)	41:(3,4,3). 5:[(161,241,161),(240,320,240)] (512000)
10:(2,1,4). 1:[(81,1,241),(160,80,320)] (512000)	42:(3,4,4). 5:[(161,241,241),(240,320,320)] (512000)
11:(1,1,4). 1:[(1,1,241),(80,80,320)] (512000)	43:(4,4,4). 5:[(241,241,241),(320,320,320)] (512000)

12:(1,2,4). 1:[(1,81,241),(80,160,320)] (512000)	44:(4,3,4). 5:[(241,161,241),(320,240,320)] (512000)
13:(2,2,4). 1:[(81,81,241),(160,160,320)] (512000)	45:(3,3,4). 5:[(161,161,241),(240,240,320)] (512000)
14:(2,2,3). 1:[(81,81,161),(160,160,240)] (512000)	46:(3,3,3). 5:[(161,161,161),(240,240,240)] (512000)
15:(1,2,3). 1:[(1,81,161),(80,160,240)] (512000)	47:(4,3,3). 5:[(241,161,161),(320,240,240)] (512000)
16:(1,3,3). 2:[(1,161,161),(80,240,240)] (512000)	48:(4,2,3). 6:[(241,81,161),(320,160,240)] (512000)
17:(1,3,4). 2:[(1,161,241),(80,240,320)] (512000)	49:(4,2,4). 6:[(241,81,241),(320,160,320)] (512000)
18:(1,4,4). 2:[(1,241,241),(80,320,320)] (512000)	50:(4,1,4). 6:[(241,1,241),(320,80,320)] (512000)
19:(1,4,3). 2:[(1,241,161),(80,320,240)] (512000)	51:(4,1,3). 6:[(241,1,161),(320,80,240)] (512000)
20:(2,4,3). 2:[(81,241,161),(160,320,240)] (512000)	52:(3,1,3). 6:[(161,1,161),(240,80,240)] (512000)
21:(2,4,4). 2:[(81,241,241),(160,320,320)] (512000)	53:(3,1,4). 6:[(161,1,241),(240,80,320)] (512000)
22:(2,3,4). 2:[(81,161,241),(160,240,320)] (512000)	54:(3,2,4). 6:[(161,81,241),(240,160,320)] (512000)
23:(2,3,3). 2:[(81,161,161),(160,240,240)] (512000)	55:(3,2,3). 6:[(161,81,161),(240,160,240)] (512000)
24:(2,3,2). 3:[(81,161,81),(160,240,160)] (512000)	56:(3,2,2). 7:[(161,81,81),(240,160,160)] (512000)
25:(1,3,2). 3:[(1,161,81),(80,240,160)] (512000)	57:(3,2,1). 7:[(161,81,1),(240,160,80)] (512000)
26:(1,3,1). 3:[(1,161,1),(80,240,80)] (512000)	58:(3,1,1). 7:[(161,1,1),(240,80,80)] (512000)
27:(2,3,1). 3:[(81,161,1),(160,240,80)] (512000)	59:(3,1,2). 7:[(161,1,81),(240,80,160)] (512000)
28:(2,4,1). 3:[(81,241,1),(160,320,80)] (512000)	60:(4,1,2). 7:[(241,1,81),(320,80,160)] (512000)
29:(1,4,1). 3:[(1,241,1),(80,320,80)] (512000)	61:(4,1,1). 7:[(241,1,1),(320,80,80)] (512000)
30:(1,4,2). 3:[(1,241,81),(80,320,160)] (512000)	62:(4,2,1). 7:[(241,81,1),(320,160,80)] (512000)
31:(2,4,2). 3:[(81,241,81),(160,320,160)] (512000)	63:(4,2,2). 7:[(241,81,81),(320,160,160)] (512000)

After balancing, many quanta have been moved. Most of the unloaded section of the problem has been allocated to one processor (id=7).

KeLP FloorPlan: Processors: 8, Q: 8, Size: 64, Shape: (4,4,4)
id:(3D partition coordinate) processor owner:[quanta region extents] (size)

0:(1,1,1). 0:[(0,0,0),(81,81,81)] (551368)	32:(3,4,2). 7:[(160,240,80),(241,321,161)] (551368)
1:(1,2,1). 0:[(0,80,0),(81,161,81)] (551368)	33:(3,4,1). 7:[(160,240,0),(241,321,81)] (551368)
2:(2,2,1). 1:[(80,80,0),(161,161,81)] (551368)	34:(3,3,1). 7:[(160,160,0),(241,241,81)] (551368)
3:(2,1,1). 1:[(80,0,0),(161,81,81)] (551368)	35:(3,3,2). 7:[(160,160,80),(241,241,161)] (551368)
4:(2,1,2). 2:[(80,0,80),(161,81,161)] (551368)	36:(4,3,2). 7:[(240,160,80),(321,241,161)] (551368)
5:(2,2,2). 2:[(80,80,80),(161,161,161)] (551368)	37:(4,3,1). 7:[(240,160,0),(321,241,81)] (551368)
6:(1,2,2). 3:[(0,80,80),(81,161,161)] (551368)	38:(4,4,1). 7:[(240,240,0),(321,321,81)] (551368)
7:(1,1,2). 3:[(0,0,80),(81,81,161)] (551368)	39:(4,4,2). 7:[(240,240,80),(321,321,161)] (551368)
8:(1,1,3). 4:[(0,0,160),(81,81,241)] (551368)	40:(4,4,3). 7:[(240,240,160),(321,321,241)] (551368)
9:(2,1,3). 4:[(80,0,160),(161,81,241)] (551368)	41:(3,4,3). 7:[(160,240,160),(241,321,241)] (551368)
10:(2,1,4). 5:[(80,0,240),(161,81,321)] (551368)	42:(3,4,4). 7:[(160,240,240),(241,321,321)] (551368)
11:(1,1,4). 5:[(0,0,240),(81,81,321)] (551368)	43:(4,4,4). 7:[(240,240,240),(321,321,321)] (551368)
12:(1,2,4). 6:[(0,80,240),(81,161,321)] (551368)	44:(4,3,4). 7:[(240,160,240),(321,241,321)] (551368)
13:(2,2,4). 6:[(80,80,240),(161,161,321)] (551368)	45:(3,3,4). 7:[(160,160,240),(241,241,321)] (551368)
14:(2,2,3). 7:[(80,80,160),(161,161,241)] (551368)	46:(3,3,3). 7:[(160,160,160),(241,241,241)] (551368)
15:(1,2,3). 7:[(0,80,160),(81,161,241)] (551368)	47:(4,3,3). 7:[(240,160,160),(321,241,241)] (551368)
16:(1,3,3). 7:[(0,160,160),(81,241,241)] (551368)	48:(4,2,3). 7:[(240,80,160),(321,161,241)] (551368)
17:(1,3,4). 7:[(0,160,240),(81,241,321)] (551368)	49:(4,2,4). 7:[(240,80,240),(321,161,321)] (551368)
18:(1,4,4). 7:[(0,240,240),(81,321,321)] (551368)	50:(4,1,4). 7:[(240,0,240),(321,81,321)] (551368)
19:(1,4,3). 7:[(0,240,160),(81,321,241)] (551368)	51:(4,1,3). 7:[(240,0,160),(321,81,241)] (551368)
20:(2,4,3). 7:[(80,240,160),(161,321,241)] (551368)	52:(3,1,3). 7:[(160,0,160),(241,81,241)] (551368)
21:(2,4,4). 7:[(80,240,240),(161,321,321)] (551368)	53:(3,1,4). 7:[(160,0,240),(241,81,321)] (551368)
22:(2,3,4). 7:[(80,160,240),(161,241,321)] (551368)	54:(3,2,4). 7:[(160,80,240),(241,161,321)] (551368)
23:(2,3,3). 7:[(80,160,160),(161,241,241)] (551368)	55:(3,2,3). 7:[(160,80,160),(241,161,241)] (551368)
24:(2,3,2). 7:[(80,160,80),(161,241,161)] (551368)	56:(3,2,2). 7:[(160,80,80),(241,161,161)] (551368)
25:(1,3,2). 7:[(0,160,80),(81,241,161)] (551368)	57:(3,2,1). 7:[(160,80,0),(241,161,81)] (551368)
26:(1,3,1). 7:[(0,160,0),(81,241,81)] (551368)	58:(3,1,1). 7:[(160,0,0),(241,81,81)] (551368)
27:(2,3,1). 7:[(80,160,0),(161,241,81)] (551368)	59:(3,1,2). 7:[(160,0,80),(241,81,161)] (551368)
28:(2,4,1). 7:[(80,240,0),(161,321,81)] (551368)	60:(4,1,2). 7:[(240,0,80),(321,81,161)] (551368)
29:(1,4,1). 7:[(0,240,0),(81,321,81)] (551368)	61:(4,1,1). 7:[(240,0,0),(321,81,81)] (551368)
30:(1,4,2). 7:[(0,240,80),(81,321,161)] (551368)	62:(4,2,1). 7:[(240,80,0),(321,161,81)] (551368)
31:(2,4,2). 7:[(80,240,80),(161,321,161)] (551368)	63:(4,2,2). 7:[(240,80,80),(321,161,161)] (551368)

Appendix B: Users Guide

API

This section presents the Application Programmer Interface (API) for the Autoblocker and Autobalancer. All code is written in the C++ language, as specified by the g++ compiler version 2.96 (Linux). The components presented in this appendix operate within the KeLP parallel framework [2, 23].

Autoblocker

The Autoblocker uses cache tiling and padding techniques to make KeLP applications cache-friendly. This component is relatively transparent to the programmer, and involves using a specialized C++ loop iterator. The following code example shows the Autoblocker being used in the KeLP local computation loop of a RedBlack3D application.

```
void ComputeLocal(IrregularGrid3 & U, const int color,
                 IrregularGrid3 & rhs)
{
    const int RED = 0 , BLK = 1;

    for (BlockedNodeIterator3 ni(U); ni; ++ni) {
        int i = ni();
        Grid3<double>& UG = U(i);

        // The full padded region.
        FortranRegion3 fullFU(UG.fullRegion());

        // The region to compute over is blocked for cache
        // Automatically by the iterator.
        FortranRegion3 FU(ni.block());

        if (color == RED)
            f_rb7rrelax(UG.data(), FORTRAN_REGION3(fullFU),
                      FORTRAN_REGION3(FU), rhs(i).data());
        else
            f_rb7brelax(UG.data(), FORTRAN_REGION3(fullFU),
                      FORTRAN_REGION3(FU), rhs(i).data());
    }
}
```

Note the use of the *BlockedNodeIterator* in the for-loop. Each iteration, the call to *ni.block()* gives the current cache tile (a subset of the problem domain) to compute over. The *fullRegion()* includes the padding elements, which are not used in the actual computation. The actual RedBlack algorithm is implemented in the Fortran routines *f_rb7*relax()*.

Autobalancer

The Autobalancer uses a Hilbert Space-Filling Curve to coordinate data-motion through a parallel cluster at runtime, to dynamically load balance an application. This component is also relatively transparent to the programmer, and again involves the *BlockedNodeIterator*. All quanta timings are obtained automatically by the *BlockedNodeIterator*, which times the activity within each loop iteration. The following code sample shows the Autobalancer being used by the RedBlack3D application.

```
void ComputeLocal(IrregularGrid3 & U, const int color,
                 IrregularGrid3 & rhs, const int radius)
{
    const int RED = 0 , BLK = 1;

    // To make the NodeIterator long-lived (outside the loop).
    BlockedNodeIterator3 ni(U);
    while (ni) {
        int i = ni();
        Grid3<double>& UG = U(i);

        // The full padded region.
        FortranRegion3 fullFU(UG.fullRegion());

        // The region to compute over is blocked for cache
        // Automatically by the iterator.
        FortranRegion3 FU(ni.block());

        if (color == RED)
            f_rb7rrelax(UG.data(), FORTRAN_REGION3(fullFU),
                      FORTRAN_REGION3(FU), rhs(i).data(), &radius);
        else
            f_rb7brelax(UG.data(), FORTRAN_REGION3(fullFU),
                      FORTRAN_REGION3(FU), rhs(i).data(), &radius);

        // Always increment the iterator at the end of the loop.
        ++ni;
    }
}
```

```

int moved;
moved=U.balance(ni.balance());
// A fast quanta mover for constant XArrays.
rhs.moveConst(ni.balance(), 0.0);
}

```

Since each iteration is timed, and the timings will dictate the balancer's actions, the application should perform all significant computation within the loop, and no more. Specifically, no I/O or other blocking calls should be executed within this loop. Since *NodeIterator*-derived loops such as the *BlockedNodeIterator* are typically the central processing loop for KeLP applications, most programs will be able to use the Autobalancer with minimal changes.

Some care must be given to the placement of the *balance()* data-transfer call. Since this function may change the location of data in the cluster, it may not occur inside the computation loop. If data was changed inside the loop, a processor could attempt to access data that has been moved off the node during load balancing. Therefore, we always call *balance()* after the *BlockedNodeIterator* loop when using the Autobalancer.

The *moveConst()* call (move constant quanta) is analogous to *balance()* but does not involve any communication. If the moved quanta moved are identical, as in the case of the RHS array above, *moveConst()* is applicable, and will run much faster than *balance()*. The key difference between the two is that *balance()* sends actual data between nodes during balancing, while *moveConst()* only allocates and deletes quanta as necessary, filling newly-created quanta with a constant value.

Quantizing a FloorPlan

Although the Autoblocker can operate on any type of floorplan, the load balancer requires a Quantization parameter as input during floorplan creation. The value of Q describes how many problem quanta to assign to a processor. As discussed in the thesis body, the higher the value of Q, the more accurately we can balance a problem, but the more communication overhead we will incur.

The following code shows an example of using the Q-Dock partitioning library to create a regular decomposition (where all quanta have identical size and shape) for a parallel RedBlack 3D problem.

```
// The number of processors.
int nP = mpNodes();

int Quantums = nP * Q;
Processors3 P(Quantums);
P.Map(HILBERT, Q);
OUTPUT(P << endl);

// Our problem region is always cubic with side length = N.
Decomposition3 T(N,N,N);

// Change this for overpartitioner.
T.distribute(BLOCK2, BLOCK2, BLOCK2, P);
OUTPUT(T << endl);

T.addGhost(1);
T.pad(true);
T.balance(false);
Manhattan grid(T);
IrregularGrid3 rhs(T);
```

The `Processors3` object decomposes the N^3 problem into a `Decomposition3 FloorPlan` with $P * Q$ quanta, where P is the number of processors, and Q is the number of quanta per processor. The `FloorPlan T` is used to create two KeLP XArrays: *grid* and *rhs*, used in the `RedBlack3D` application. The `Decomposition` class derives from `FloorPlan`, and the `Manhattan` and `IrregularGrid` classes derive from `XArray`.

The `pad()` and `balance()` method calls switch on or off Padding for cache and Load Balancing features respectively. Without padding, the `Autoblocker` will still tile the problem, but will not add dummy padding elements to the data arrays. When the balancer is turned on, the potentially-expensive *Timing Publish* communication phase will occur automatically when a *BlockedNodeIterator* is used to coordinate computation. These features will affect memory overhead and communication time respectively, so use them with care.

Code Design

In this section we present the relevant C++ classes used in the Autoblocker and Autobalancer, and describe their function.

Kelp/Kelp Library

These classes are located in the central kelp library libkelp.so.

- *ShapedFloorPlanX* – An X-dimensional FloorPlan that knows the processor region. This contrasts to the normal KeLP FloorPlan that views each partition as elements of a 1D array. The processor region describes how processors are oriented relative to each other in the problem domain. The Q-Dock library ensures the processor region is as square as possible to minimize inter-processor surface area (and therefore communication).
- *TimingsX* – An X-dimensional array of quanta timings. This array is the same shape as the processor region. Uses MPI broadcasts to implement the Timing Publish phase of the load balancer. Also performs statistics on timings to remove outlier measurements.
- *BalanceX* – A base class that coordinates the balancing algorithm. The XArray has an instance of this class, and so it is persistent across *BlockedNodeIterator* loops. BalanceX tracks the FloorPlan of the problem, and ensures that is up to date in the face of changes. Has a TimingsX instance.
- *RCBX* – A derived class of BalanceX. Implements the Recursive-Coordinate-Bisection algorithm using timings from its parent BalanceX class.

Kelp/Q-Dock Library

These classes comprise the new Quantum Decomposition (Q-Dock) library, which is derived from the KeLP 1.3 Dock library.

- *BlockedNodeIteratorX* – A 2 or 3-dimensional class, derived from *ParallelIterator*. Used in the place of *NodeIterator*, it implements the Autoblocker and Autobalancer functionality. Has the ability to collect quanta timings transparently using the *MPI_Wtime()* wall-clock function.
- *ProcessorX* – A X-dimensional class that knows how to decompose a parallel problem using a Quantization factor, and how to order quanta along a Hilbert Space-Filling Curve.

REFERENCES

1. J. R. Pilkington and S. B. Baden. *Dynamic partitioning of non-uniform structured workloads with spacefilling curves*. IEEE Transactions on Parallel and Distributed Systems, 7(3):288--300, Mar. 1996. 37
2. S. Baden and S. Fink. *The Data Mover: A machine independent abstraction for managing customized data motion*. LCPC, August 1999.
3. S. J. Fink, S. R. Kohn, and S. B. Baden. *Efficient run-time support for irregular block-structured applications*. Journal of Parallel and Distributed Computing, 50:61--82, May 1998.
4. C. Rivera and C.-W. Tseng, *Data transformations for eliminating conflict misses*, in Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation, 1998, pp. 38--49.
5. S. Coleman and K. McKinley. *Tile size selection using cache organization and data layout*. In Proc. Programming Language Design and Implementation, pages 279-290, 1995.
6. G. Rivera and C.-W. Tseng. *A comparison of compiler tiling algorithms*. In 8th International Conference on Compiler Construction (CC'99), March 1999.
7. G. Rivera and C.-W. Tseng. *Tiling optimizations for 3d scientific computations*. In Proceedings of SC'00, Dallas, TX, November 2000.
8. Diekmann, R., Monien, R., Preis, R *Load Balancing Strategies for Distributed Memory Machines*. Parallel and Distributed Processing for Computational Mechanics: Systems and Tools, pages 124-157, Saxe-Coburg 1999.
9. B. Hendrickson and T. G. Kolda. *Graph partitioning models for parallel computing*. Parallel Computing, 26:1519--1534, 2000.
10. G. Cybenko. *Dynamic load balancing for distributed memory multiprocessors*. J. of Parallel Distributed Computing , 7:279--301, 1989.
11. R. H. Arpaci, D. E. Culler, A. Krishnamurthy, S. G. Steinberg, and K. Yelick. *Empirical evaluation of the CRAYT3D: A compiler perspective*. In Proceedings of the 22nd Annual International Symposium on Computer Architecture, pages 320--331, Santa Margherita Ligure, Italy, June 1995.
12. R. Diekmann, A. Frommer, and B. Monien. *Efficient schemes for nearest neighbor load balancing*. Parallel Computing, 25:789--812, 1999.

13. D. H. Gibson. *Considerations in block-oriented systems design*. AFIPS Conference Proceedings, 30; SJCC, 75-80, 1967.
14. M. D. Hill. *Aspects of Cache Memory and Instruction Buffer Performance*. Ph.D. Thesis, UCB, Technical Report UCB/CSD 87/381, November 1987.
15. A. J. Smith. *Cache memories*. Computing Surveys 14:3, 473-530, September 1982.
16. J. L. Hennessy, D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Second Edition. Morgan Kaufmann Publishers, Inc. San Francisco, CA, 1996.
17. M. Lam, E. Rothberg, and M. E. Wolf. *The cache performance and optimizations of blocked algorithms*. Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV), Santa Clara, CA, April 1991.
18. S. B. Baden, R. B. Frost, D. Shalit. *KeLP User Guide Version 1.3*. CSE Department, University of California, San Diego, September 1999.
19. J. S. Liptay. *Structural aspects of the System/360 Model 85, Part II: The Cache*. IBM Systems Journal. 7:1, 15-21, 1968.
20. C. L. Lawson, R. J. Hanson, D. Kincaid, and F. T. Krogh. *Basic Linear Algebra Subprograms for FORTRAN usage*. ACM Trans. Math. Software, 5:308--323, 1979.
21. David F. Bacon, Susan L. Graham, and Oliver J. Sharp. *Compiler transformations for high-performance computing*. ACM Computing Surveys, 26(4), 1994.
22. N. Mitchell, L. Carter, J. Ferrante, and K. Hogstedt. *Quantifying the multi-level nature of tiling interactions*. In Proceedings of the Tenth Workshop on Languages and Compilers for Parallel Computing, Minneapolis, MN, August 1997.
23. S. J. Fink. *A Programming Model for Block-Structured Scientific Calculations on SMP Clusters*. UCSD CSE Department, Ph.D Dissertation, June 1998.
24. K. Gatlin and L. Carter. *Architecture-cognizant divide and conquer algorithms*. In Proceedings of SC'99, Portland, OR, November 1999.
25. S. VanderWiel, and D. Lilja. (1997). *When Caches Aren't Enough: Data Prefetching Techniques*. IEEE Computer, 30(7):23—30
26. AMD. *AMD Athlon Processor Model 4 Data Sheet*. http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/23792.pdf, November 2001.

27. P. M. Papadopoulos, M. J. Katz, G. Bruno. *NPACI Rocks: Tools and Techniques for Easily Deploying Manageable Linux Clusters*. Accepted and to appear in: Cluster 2001, Newport Beach, October, 2001.
28. Intel Corporation. *Intel Pentium III Processor Overview*. <http://developer.intel.com/design/pentiumiii/prodbref/index.htm>
29. San Diego Supercomputing Center (SDSC). *Blue-Horizon: NPACI TeraFlops IBM SP*. <http://www.sdsc.edu/Resources/bluehorizon.html>.
30. S. B. Baden, S. Fink. *A Programming Methodology for Dual-tier Multicomputers*. IEEE Transactions on Software Engineering, 26(3): 212-26, March 2000.
31. A. Brandt, *Guide to multigrid development*, in Multigrid Methods, W. Hackbusch and U. Trottenberg, eds., Lecture Notes in Mathematics 960, Springer, Berlin/Heidelberg, 1982.
32. F. C. H. Lin and R. M. Keller. The Gradient Model Load Balancing Method. *IEEE Transactions on Software Engineering*, SE-13(1):32--38, January 1987.
33. R. P. R. Diekmann, B. Monien. *Load balancing strategies for distributed memory machines*. In Karsch/Monien/Satz, editor, Multi-Scale Phenomena and their Simulation, pages 255--266. World Scientific, 1997.
34. V. Kumar, A. Grama, and V. Rao. *Scalable load balancing techniques for parallel computers*. Journal of Parallel and Distributed Computing, 22(1), 1994.
35. Christian Weiss, Markus Kowarschik, Ulrich Ruede, and Wolfgang Karl. *Cache-Aware multigrid methods for solving poisson 's equation in two dimensions*, 1999. <http://www.mgnet.org/mgnet/Conferences/ParMGM98/Papers/kowarschik.html>
36. M. Ripeanu, A. Iamnitchi. And I. Foster. *Performance Predictions for a Numerical Relativity Package in Grid Environments*. International Journal of Scientific Applications, 14 (4). 2001.
37. B. Hendrickson and R. Leland. *A multilevel algorithm for partitioning graphs*. In Proceedings of Supercomputing 95, December 1995.
38. B. Hendrickson and K. Devine. *Dynamic load balancing in computational mechanics*. Computer Methods in Applied Mechanics and Engineering, 184:485-500, 2000.
39. R. D. Williams. *Performance of dynamic load balancing algorithms for unstructured mesh calculations*. Concurrency: Practice & Experience, 3:457--481, 1991.

40. V. Kumar, A. Grama, V. Rao. 1994. *Scalable load balancing techniques for parallel computers*. Journal of Parallel and Distributed Computing, 22(1):60-79.
41. B. Hendrickson and T. G. Kolda. *Graph partitioning models for parallel computing*. Parallel Computing, 26:1519--1534, 2000.
42. S. T. Barnard and H. D. Simon. *A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems*. Concurrency: Practice & Experience, 6(2):101--117, 1994.
43. H. D. Simon. *Partitioning of unstructured problems for parallel processing*. Computing Systems in Engineering, 2(2/3):135--148, 1991.
44. M.J. Berger and S. Bokhari. *A partitioning strategy for non-uniform problems on multiprocessors*. IEEE Trans. Computers, C-26:570--580, 1987.
45. R. Wolski. *Dynamically forecasting network performance to support dynamic scheduling using the network weather service*. In Proc. 6th IEEE Symp. on High Performance Distributed Computing, August 1997.
46. SDSC. *NPACI / NSF Distributed Terascale Facility (DTF)*. http://www.sdsc.edu/Press/01/080901_teragrid.html
47. IBM Corporation. *IBM Redbooks | RS/6000 Scientific and Technical Computing: POWER3 Introduction and Tuning Guide* <http://www.redbooks.ibm.com/pubs/pdfs/redbooks/sg245155.pdf>
48. Kyle Gallivan, William Jalby, Ulrike Meier, and Ahmed H. Sameh. *Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design*. The International Journal of Supercomputer Application, 2(1):12--48, 1988.
49. D. Gannon, W. Jalby, and K. Gallivan. *Strategies for cache and local memory management by global program transformations*. Journal of Parallel and Distributed Computing, 5:587-616, 1988.
50. G. Karypis, K. Schloegel, and V. Kumar. *Parmetis---parallel graph partitioning and sparse matrix ordering library*, version 2.0. Univ. of Minnesota, Minneapolis, MN, 1998.
51. A. Silberschatz, P. B. Galvin. *Operating System Concepts*. Fifth Edition. Addison-Wesley, 1998.
52. C. L. Bottasso, J. E. Flaherty, C. Ozturan, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. *The quality of partitions produced by an iterative load balancer*. In B. K. Szymanski and B. Sinharoy, editors, Proceedings Third

Workshop on Languages, Compilers, and Runtime Systems, Kluwer, Boston, pp. 265--277, 1995.

53. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
54. Manish Parashar and J.C. Browne. *On partitioning dynamic adaptive grid hierarchies*. In 29th Annual Hawaii International Conference on Systems Sciences, pages 604-613, Maui, Hawaii, January 1996.
55. J. Teresco, M. Beall, J. Flaherty, and M. Shephard. *A Hierarchical partition model for adaptive finite element computation*. Technical report, Dept. of Computer Science, Rensselaer Polytechnic Institute, 1998.
56. F. Irigoien and R. Triolet. *Supernode partitioning*. In 15th Annual ACM Symposium on Principles of Programming Languages, pages 319--329, San Diego, California., Jan. 1988.