# Accounting for Memory Bank Contention and Delay in High-Bandwidth Multiprocessors *

Guy E. Blelloch * Phillip B. Gibbons †
Yossi Matias † Marco Zagha *

* School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3891
{blelloch,marcoz}@cs.cmu.edu

† AT&T Bell Laboratories
600 Mountain Avenue
Murray Hill, NJ 07974
{gibbons,matias}@research.att.com

**Abstract**

For years, the computation rate of processors has been much faster than the access rate of memory banks, and this divergence in speeds has been constantly increasing in recent years. As a result, several shared-memory multiprocessors consist of more memory banks than processors. The object of this paper is to provide a simple model (with only a few parameters) for the design and analysis of irregular parallel algorithms that will give a reasonable characterization of performance on such machines. For this purpose we extend Valiant's bulk-synchronous parallel (BSP) model with two parameters: a parameter for memory bank *delay*, the minimum time for servicing requests at a bank, and a parameter for memory bank *expansion*, the ratio of the number of banks to the number of processors. We call this model the $(d, \mathbf{x})$-BSP. We show experimentally that the $(d, \mathbf{x})$-BSP captures the impact of bank contention and delay on the CRAY C90 and J90 for irregular access patterns, without modeling machine-specific details of these machines. The model has clarified the performance characteristics of several unstructured algorithms on the CRAY C90 and J90, and allowed us to explore tradeoffs and optimizations for these algorithms. In addition to modeling individual algorithms directly, we also consider the use of the $(d, \mathbf{x})$-BSP as a bridging model for emulating a very high-level abstract model, the Parallel Random Access Machine (PRAM). We provide matching upper and lower bounds for emulating the EREW and QRQW PRAMs on the $(d, \mathbf{x})$-BSP.

*A preliminary version of this paper appeared in Proc. 7th ACM Symp. on Parallel Algorithms and Architectures, July 1995.

**Index terms:** Memory bank contention, memory delays, parallel machine models, performance analysis, parallel algorithms, shared memory, multiprocessors.

**Acknowledgements:** The experiments were run on the CRAY C90 at the Pittsburgh Supercomputing Center (PSC) and on a CRAY J90 at Cray Research. We are grateful to Charles Grassl at Cray and Raghurama Reddy at the PSC for their help running the experiments and getting us exclusive access to the machines. Thanks to Max Dechantsreiter at Tera and Patrick McGehearty at Convex for useful information on the Tera and Convex. Thanks to J. for the papers. The comments of the anonymous referees were helpful in improving the presentation of this paper.

This research was supported in part by the Defense Advanced Research Projects Agency (DARPA) under grant number F33615-93-1-1330 and in part by an NSF Young Investigator Award.

# 1 Introduction

In recent years several models have been designed with the goal of abstracting away from details of parallel machines while still giving guidance in developing efficient algorithms. Examples of such models include the Bulk Synchronous Parallel (BSP) [52] and LOGP [16] models, which both aim to serve as high-level performance models of message-passing machines. The important feature of these two models is that they abstract away from many details of the network, including topology, while still capturing important aspects such as bandwidth and latency. The purpose of the models is to help in optimizing algorithms, to aid in understanding how algorithm scale, and to give guidance in choosing among algorithms, all without needing to consider details of the machine. They are often used in conjunction with experimentation to account for aspects that are not considered by the models, such as local computation times. As such, they have been quite successful, leading to practical designs of various algorithms [3, 6, 14, 16, 17, 22, 25, 29, 37].

In this paper we introduce and evaluate a model with similar goals, but for shared-memory machines instead of message-passing machines. Due to the wide variety of shared-memory machines, we limit ourselves both in terms of the class of machines and the class of algorithms we consider. In particular we are concerned with the class of machines that (1) have a high-bandwidth network between the processors and memory banks, (2) allow for fine-grained memory accesses, (3) have memory banks that are slower than the processors and compensate by having more memory banks than processors, and (4) can tolerate latency to the memory from processors by allowing for multiple outstanding memory requests. Such machines include both vector multiprocessors, such as the CRAY C90 and J90, and multithreaded machines, such as the TERA MTA [2]. There is also some evidence that Symmetric multiprocessors (SMPs) are converging on these features, with increased bandwidth and better latency tolerating techniques.

This work was motivated by the study of algorithms with irregular memory access patterns, such as sorting [54], sparse-matrix vector product [7] and graph algorithms [26, 53], on the CRAY C90. In our analysis we found previous models either quite detailed, or inadequate for describing the key performance characteristics of the algorithms. For example, we found that a straightforward shared-memory variant of the BSP does not properly account for contention at the banks since there is no way to account for relative speed of memory banks and processors. On the other hand, previous models of multibank memory systems [5, 9, 13, 38, 15, 4, 11, 10, 46, 27, 18, 51, 47, 48] are highly detailed, and the studies have only considered either regular or random access patterns. In this paper we are interested in modeling algorithms with irregular, but not necessarily random, access patterns without requiring a complicated model. We are particularly concerned with capturing the effect of memory contention in these algorithms. We assume that the

| Machine | Procs ($p$) | Banks ($B$) | Expansion ($\mathbf{x}$) |
|---|---|---|---|
| NEC SX-3 | 4 | 1024 | 256 |
| Tera MTA | 256 | $512 \times 64$ | 128 |
| Cray C90 | 16 | 1024 | 64 |
| Cray J90 | 16 | $256 \times 2$ | 32 |
| Convex C4 | 4 | 128 | 32 |
| Meiko CS-2 node | 2 | 16 | 8 |
| SGI Power Challenge | 18 | $8 \times 8$ | 3.5 |

Table 1: Expansion in the number of memory banks for various machines. In the current Tera design, memory banks are organized into memory modules containing 64 memory banks. The Cray J90 memory banks are organized in pairs that share address and data paths. The SGI Power Challenge memory is arranged as up to 8 modules with 8 banks each. The Meiko CS-2 node contains two Fujitsu $\mu$-VP vector processors.

parts of applications with regular memory access patterns can be analyzed with more traditional approaches [15, 27].

The model we consider is a shared-memory variant of the BSP. It is based on assuming a set of processors are connected through a pipelined interconnection network to a set of memory banks. In addition to the three parameters of the BSP—the number of processors $p$, the *throughput* (bandwidth) parameter $g$, and the *latency* parameter $L$—our model has two additional parameters—the *bank delay* $d$, and the *expansion factor* $\mathbf{x}$. The bank delay is the throughput at a memory bank, and the expansion factor is the ratio of the number of memory banks to the number of processors. Table 1 lists several machines along with their expansion factor. Our experiments show that these additional parameters are necessary for taking account of high contention on the CRAY C90, where the best sustainable gap, $g$, between memory requests issued by each individual processor is less than the best sustainable delay, $d$, between accesses to each individual memory bank. As with the BSP we assume a set of bulk synchronous steps. During each step the processors can execute either local operations or accesses to the global memory, but the memory is not guaranteed to be coherent until the beginning of the next step. We denote this model as the $(d, \mathbf{x})$-BSP (the "deluxe" BSP). Based on the $(d, \mathbf{x})$-BSP model we show a number of results, both experimental and theoretical.

We first show experimentally that the $(d, \mathbf{x})$-BSP can predict the performance of the CRAY vector multiprocessors with fairly good accuracy in many situations with irregular access patterns, even though the model ignores details of the machines.[1] We show this

[1] We assume, however, that the code is mostly or fully vectorized so that memory traffic is high. We

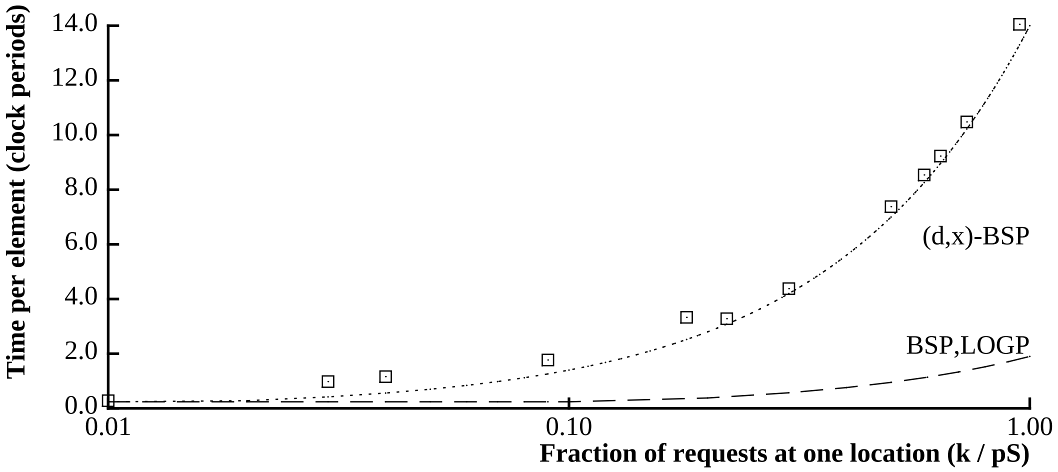



Figure 1: Predicted and measured performance on a set of memory access patterns extracted from a trace of Greiner's algorithm for finding the connected components of a graph [26]. Measured times on an 8 processor CRAY J90 for several patterns are shown with squares. Predicted times are given for a direct shared-memory variant of the BSP and for the $(d, \mathbf{x})$-BSP as a function of contention.

both for the CRAY C90, which uses static RAM (SRAM) and has a bank delay of 6 clock cycles, and for the more modestly priced CRAY J90, which uses dynamic RAM (DRAM) and has a bank delay of 14 clock cycles. The $(d, \mathbf{x})$-BSP has made it easier to analyze several algorithms and has allowed us to predict various effects that cannot be predicted without taking into account the bank delay. For example, Figure 1 and later Figure 13 compare predicted and measured times for an implementation of a graph-connectivity algorithm [26] and a sparse-matrix vector multiplication [7] algorithm. These times are compared with the predictions based on a direct shared-memory variant of the BSP that does not account for the bank delay. Although the $(d, \mathbf{x})$-BSP does not model the time exactly, due mostly to ignoring effects of the network, it more accurately captures the effect of high contention. Furthermore, the discrepancy between the BSP and $(d, \mathbf{x})$-BSP becomes larger as either the bank delay or the number of processors increases (the experiment shown is for only 8 processors).

Second, we study to what extent the effects of multiple memory locations residing in a single bank can be ignored, when using random mappings of memory locations to memory banks. Many researchers have studied the effect of randomly mapping memory to banks (*e.g.* [36, 32, 41, 52, 2, 40, 42, 35, 28]). If there is sufficient parallel "slackness" (extra parallelism) so that each bank is receiving multiple requests, it has been shown [36, 32, 41, 52] that with high probability the memory references will be reasonably balanced across the banks. These results, however, assume that there is no contention at individual

do not claim that the model applies to programs that have large scalar components.

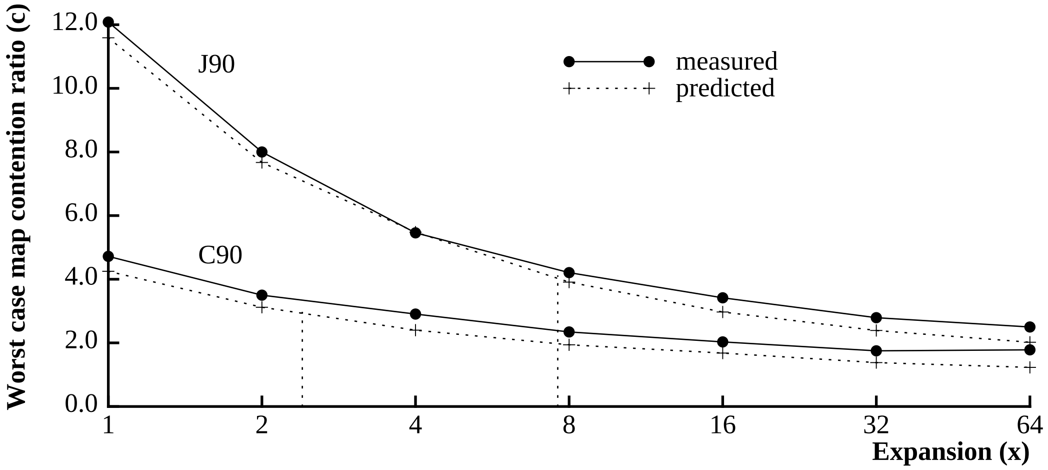



Figure 2: The ratio of the time that includes the effect of multiple memory locations being mapped to the same bank to the time that excludes the effect, when using random mapping. This is given as a function of expansion and is for a worst-case reference pattern. For both the CRAY C90 and J90 the actual expansion is 64 and the other data points are taken by using a subset of the banks. This shows the advantage of having an expansion greater than the $d/g$ (shown with the dotted vertical lines) that is needed to match the servicing bandwidth at the banks with the issuing bandwidth at the processors.

memory locations.[2] If we allow for contention at memory locations then ignoring the mapping of memory locations to banks can significantly underpredict the running time even for unbounded slackness. This is because even when the number of memory requests is large, the number of *accessed* locations could be small and not well balanced across the memory banks. We study the impact of the expansion factor on the balance of requests, when only a small number of memory locations are accessed, and show that increasing the expansion factor reduces the effect of ignoring the mapping of memory locations to banks. For the CRAYC90, which has a high expansion factor, we show both experimentally and analytically that ignoring the mapping will underpredict running time by a factor of about 2.0 for a worst case reference pattern and typically by much less. The effect of expansion for the CRAY C90 and CRAY J90 is shown in Figure 2.

Third, we explore scenarios under which two very high-level models for algorithm design, the EREW PRAM (*e.g.* [31]) and the stronger QRQW PRAM [24], can be effectively mapped onto high-bandwidth machines (small $g$) when properly accounting for memory bank delay. For the case $\mathbf{x} < d/g$, we observe that $(d/\mathbf{x})$ is an inevitable work overhead due to the insufficient bandwidth at the memory banks, and provide an emulation of the QRQW PRAM on the $(d, \mathbf{x})$-BSP in which the overhead matches this factor. For the case

[2] In the case of Ranade's work [41] it is assumed that references to a single location are combined in the network.

$\mathbf{x} \geq d/g$, we present a work-preserving emulation of the QRQW PRAM on the $(d, \mathbf{x})$-BSP, assuming $g$ is a small constant, where the effect of $d$ on the slowdown of the emulation is partially compensated for by the expansion factor $\mathbf{x}$. These two emulations assume a random mapping of memory locations to memory banks, but often it suffices to randomly order the data at the beginning of an algorithm. These PRAM emulations on the $(d, \mathbf{x})$-BSP generalize the PRAM emulations on the BSP given in [52, 23].

Finally, we experiment with four algorithms with irregular memory access patterns: a QRQW binary search algorithm, a QRQW random permutation algorithm, a sparse matrix multiply, and a CRCW connected components algorithm. The $(d, \mathbf{x})$-BSP model is used to predict the running time of each algorithm.

# 2 Accounting for memory bank contention and delay

Our model is an extension of Valiant's BSP model [52]. The BSP model was introduced to be a "bridging" model between software and hardware in parallel machines: software would be designed for this model and parallel machines would implement it. Such a standardized interface would allow software to be more easily ported to various hardware platforms. The BSP model consists of $p$ processor/memory components communicating by sending point-to-point messages. The interconnection network supporting this communication is characterized only by a throughput parameter $g$ and a latency parameter $L$. The particular topology of the network is ignored and the cost to communicate among processors is assumed to be uniform, independent of the identity of the processors. A BSP computation consists of a sequence of "supersteps" separated by bulk synchronizations (typically, a barrier synchronization among all the processors). In each superstep the processors can perform local computations and send and receive a set of messages. Messages are sent in a pipelined fashion (i.e., each processor may issue messages and continue with its computation prior to the receipt of those messages). Messages sent in one superstep will arrive prior to the start of the next superstep. The time charged for a superstep is calculated as follows. Let $W_i$ be the amount of local work performed by processor $i$ in a given superstep. Let $S_i$ ($R_i$) be the number of messages sent (received) by processor $i$. Let $W = \max_{i=1}^{p} W_i$, $S = \max_{i=1}^{p} S_i$, and $R = \max_{i=1}^{p} R_i$. Then the cost, $T$, of a superstep is defined to be

$$T = \max(W,\ g \cdot S,\ g \cdot R,\ L) \ .$$

Intuitively the communication throughput parameter, $g$, is the best sustainable gap between message sends issued by each individual processor; therefore $1/g$ represents the available bandwidth per processor. Intuitively the communication latency parameter,

$L$, called the "periodicity factor" in [52], is the worst case time to deliver a message between two processors in an otherwise unloaded network plus the time to perform a barrier synchronization.

The $(d, \mathbf{x})$-BSP differs from the BSP described above as follows. The $(d, \mathbf{x})$-BSP is an explicit shared-memory model. Memory components (banks) are considered as separate from the processors, and their number is accounted for in the model. Instead of sending and receiving messages, processors make global memory requests which are serviced directly by the memory banks. To account for differences in speed of memory requests by processors and responses by memory banks the $(d, \mathbf{x})$-BSP also assigns a distinct throughput parameter for the memory banks.

**The $(d, \mathbf{x})$-BSP model.** The $(d, \mathbf{x})$-BSP is depicted in Figure 3. It consists of $p$ processors communicating by reading and writing memory words from a separate set of $B$ memory banks; these memory banks are used as a shared memory by the processors. In practice these memory banks might be physically located next to the processors, but it is assumed that the processors are not involved in handling incoming memory requests. Processors also have local memory for use with local operations; this accounts for registers, cache memory, and main memory in each processor's local environment.

A $(d, \mathbf{x})$-BSP computation consists of a sequence of supersteps separated by barrier synchronizations. In each superstep the processors can perform local computations and make a set of pipelined, global memory requests. Requests made in one superstep will complete prior to the start of the next superstep. We include the same parameters as included in the BSP model but add two more: the *memory delay* and the *bank expansion factor*. The parameters of the $(d, \mathbf{x})$-BSP are summarized as follows:

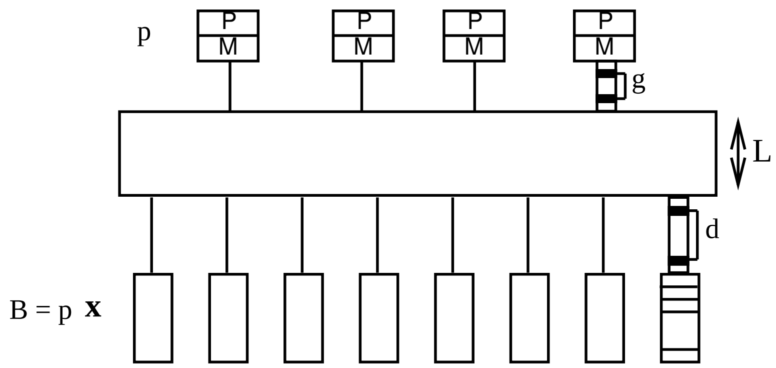



Figure 3: The $(d, \mathbf{x})$-BSP model.

| | |
|---|---|
| $p$ | number of processors |
| $g$ | communication throughput parameter (gap) |
| $L$ | the periodicity parameter (latency + synchronization) |
| $d$ | memory bank throughput parameter (delay) |
| $\mathbf{x}$ | memory bank expansion factor (assumed to be $\geq 1$) |

where the number of memory banks, denoted as $B$, is $\mathbf{x} \cdot p$. Intuitively, the gap parameter $g$ is the best sustainable gap between memory requests (either reads or writes) issued by each individual processor. Intuitively, the periodicity parameter, $L$, is the worst case time to complete a single memory read in an otherwise unloaded memory system plus the time to perform a barrier synchronization.

The time charged for a superstep is calculated as follows. Similar to the BSP, let $W$ be the maximum amount of local work performed by any one processor in the superstep, and let $S$ be the maximum number of global memory requests made (the maximum memory request load) by any one processor. Let $R_j$ be the number of requests handled by memory bank $j$, and let $R = \max_{j=1}^{\mathbf{x} \cdot p} R_j$. Then the cost, $T$, of a $(d, \mathbf{x})$-BSP superstep is defined to be

$$T = \max(W,\ g \cdot S,\ d \cdot R,\ L)\ .$$

We refer to a machine as $(d, \mathbf{x})$-*balanced* if $\mathbf{x} = d/g$. This is the point where the total bandwidth available at the processors and network for random access patterns matches the total bandwidth available at the memories. Although we have chosen to extend the BSP model, it should be straightforward to extend other related models, *e.g.* the LOGP [16] or DMM [36] models, with the $d$ and $\mathbf{x}$ parameters.

The contention at a memory bank can be due to not only the contention at a particular location in the bank, but also due to accesses to multiple locations within the bank. In the basic model we make no assumptions about how the memory locations are mapped onto the memory banks. This allows us to consider both scenarios where the mapping is under user control and where the mapping is random. To separate the two types of contentions we make the following definitions, each of which is defined relative to a single superstep. Let the *memory request contention*, $k_i$, to a location $i$ be the number of requests to $i$, and let $k = \max_i k_i$. Let $M_j$ be the set of locations mapped to memory bank $j$. The size of this set is the *module map contention*, $\mu_j = |M_j|$, of bank $j$, and let $\mu = \max_j \mu_j$. Then

$$R_j = \sum_{i \in M_j} k_i$$

is the *module load contention* of bank $j$ (see Figure 4).

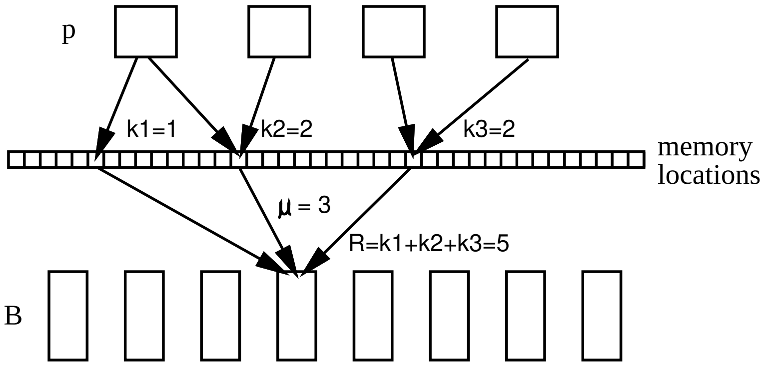



Figure 4: Memory request contention ($k1, k2, k3$), module map contention ($\mu$), and module load contention ($R$).

**Applicability of the model**. The model assumes several properties of a machine. We now discuss the scope and limitations of the model.

First, the model assumes that each processor has enough outstanding memory requests to compensate for both network latency and bank delays. Allowing for multiple outstanding requests is relatively easy for memory writes, but more difficult for memory reads. Techniques for allowing multiple outstanding reads, often called latency hiding techniques, include vectorization, multithreading, prefetching, non-blocking caches, and other methods for decoupling the request for the memory from its use. Vectorization has been used for over 20 years to hide memory latency and has the advantage that it is simple to implement. On the other hand it restricts the kinds of program that can be used. Multithreading was suggested and implemented for hiding latency on the HEP [45] and was later used in the design of the TERA and Sparcle [1]. Multithreading is more complicated to implement than vectorization but permits the use of a wider class of programs. Prefetching and non-blocking caches are becoming common on commodity processors, although the number of outstanding requests currently allowed (typically between 2 and 8 words or cache lines) probably cannot compensate for the latency to a large shared memory. If processors evolve so that they permit additional outstanding read requests, then it is quite possible that the model will apply to commodity processors attached to fast multi-bank memory systems.

Second, although the model accounts for contention at the processors and memory banks, it does not account for contention within the network (similar assumptions are also made by the BSP and LOGP models). The definition of the $g$ parameter accounts for network bandwidth under normal conditions, but in many networks it is possible to set up particularly bad permutations.[3] In fact, in the next two sections we show that for the

[3]As discussed in the next section, we derive the $g$ parameter assuming a random permutation of ad-

CRAY C90 and J90, the model breaks down under certain contrived conditions; however, under a random mapping of memory locations to banks, these bad conditions are very unlikely to occur.

Third, the model assumes that the memory banks are slower than the rate at which processors can issue memory requests into the network (*i.e.*, $d > g$). With today's memory technology and processor speeds, the $d$ parameter is typically on the order of 10 clock cycles. For the $g$ parameter to be less than that, the bandwidth into the network per processor needs to be quite high.

Fourth, we assume that in each superstep of the $(d, \mathbf{x})$-BSP, each processor injects its memory requests into the network in a random order (although in practice we find this is not necessary). This assumption is made since even if the requests are reasonably distributed among the banks within the whole superstep, they might be badly distributed over time during the superstep. If we assume infinite buffering in the network and at the banks and assume that requests can overtake each other, then this may not be a problem, but most networks have only limited buffering. The limited buffering can cause congestion that will back up future requests even though they are going to a non-congested destination. The problem can be compounded by processor-memory feedback effects [46]. Our experiments on the CRAY J90 have shown slowdowns of over a factor of 20 using bad injection orders as compared to random injection. We note, however, that in practice it is often not necessary to spend extra time randomly ordering requests since it is known that the requests are well distributed.

Fifth, the $(d, \mathbf{x})$-BSP does not explicitly model the processors' local environments, including cache behavior and local arithmetic operations. As with the BSP and LOGP models, the $(d, \mathbf{x})$-BSP focuses on the interprocessor communication aspects of parallel machines, as these are presumed to be the primary bottlenecks of parallel programs. Since the $(d, \mathbf{x})$-BSP can only model the local work within a rough estimate, experiments are needed to get an accurate measure of this component. Typically a small experiment will suffice to get an accurate prediction of work over a range of problem sizes and number of processors [16, 8].

Another consideration regarding the local environment is in accounting for the use of caches. In cache-based Symmetric multiprocessors (SMPs), understanding the cache behavior is often necessary in order to obtain reasonably accurate performance prediction. In the $(d, \mathbf{x})$-BSP it is up to the user to determine, for each shared memory reference, whether the value is present in its local cache or must be retrieved from the memory (or some other processor's cache). A local cache hit is accounted as a local operation; a cache miss is accounted as a global operation. Different machines have different cache

dresses. For certain regular permutations, the time through the network could be better than predicted.

policies, and the accuracy of the $(d, \mathbf{x})$-BSP prediction depends in part on the extent to which operations can be properly accounted as local or global. Furthermore, it might be necessary to account for memory traffic caused by the cache coherence protocol itself. Considerations of cache behavior or other uses for the local memory provided by the $(d, \mathbf{x})$-BSP do not arise in our experiments on the CRAY C90 and J90, since these machines have adequate memory bandwidth, limited local memory, and no caches for vector data.

Finally, the model does not take account of the possibility of caching at the memory banks, as available in the design of the TERA [2], and suggested by Hsu and Smith [30]. Extending the model to account for caching at the bank is an interesting area of future work.

# 3 Case study: modeling the Cray

This section presents a qualitative and quantitative comparison of the CRAY C90 and CRAY J90 to the $(d, \mathbf{x})$-BSP. The experiments in this section provide evidence that the abstract model can produce realistic predictions of running times despite ignoring many architectural details, such as the use of vector processors, the topology of the interconnection network, the flow control for the network, and the priority scheme on the banks. The experiments also show some cases where the model breaks down. The features of the CRAY vector machines that make them suitable for the model are the following:

1. The use of vector gather and scatter instructions to allow for multiple outstanding memory requests to an arbitrary set of addresses.

2. A high bandwidth interconnection network between the processors and the memory banks that supports fine-grained memory references.

3. Memory banks that are significantly slower than the rate at which processors can issue memory requests. The ratio $d/g$ is 5 on the C90 and 7.7 on the J90.

However, to use the $(d, \mathbf{x})$-BSP model for the CRAY vector machines we have to separate out the costs of regular versus irregular accesses. In particular on the CRAY since the bandwidth for unit stride accesses is very high (the bandwidth for two loads and a store is the same as for an add) and the load across the banks is perfectly distributed for such accesses, we count such unit stride accesses as part of the work $W$.

Here we discuss the features of the CRAY C90 and J90 in more detail. The instruction set supports vector load and store instructions, which load or store up to 128 words per

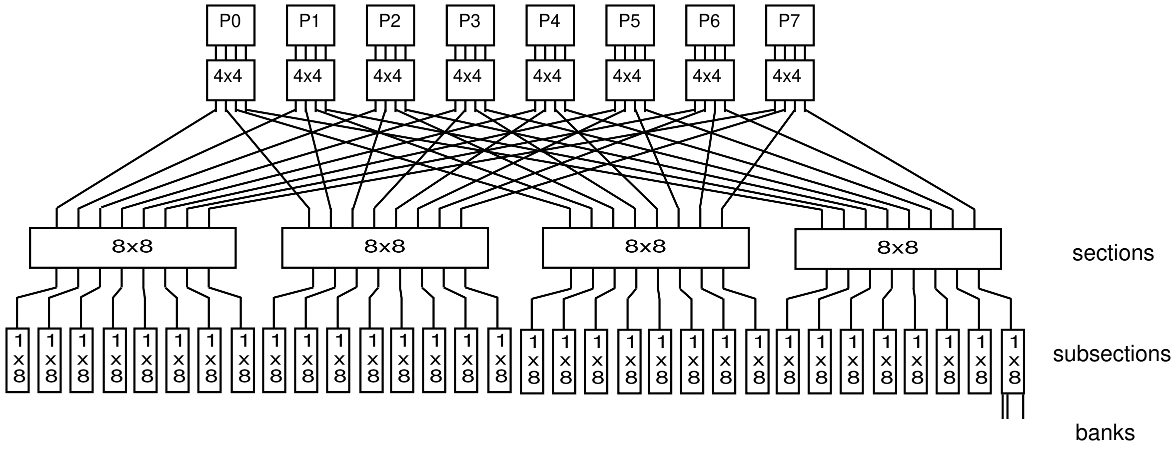



Figure 5: Cray Y-MP interconnection network (adapted from Smith and Taylor [46]).

instruction (64 on the CRAY J90). These instructions can either be strided for regular access patterns, or can be based on a vector of addresses for indirect addressing. In the indirect loads and stores, often called *gather* and *scatter*, each address specifies the location of a single 64-bit word allowing for fine-grained memory accesses. The processors are connected to the memory banks with a multistage network (see Figure 5). In the largest memory configuration of the CRAY C90, banks are divided into 8 "sections" at the first level, which are each further subdivided into 8 "subsections", each of which contains 16 memory banks. Each processor has independent paths through the section and subsection, but processors can interfere with each other due to bank conflicts. Sequential memory locations are interleaved across memory banks, making regular, strided memory access fast (except on strides which are large powers of two).

The latency through the network is about 9 clock cycles each way. When added to the access time at the memory bank, the total latency for a load is between 23 and 35 clock cycles, depending on the CRAY model and assuming no contention at the memory bank. This latency is usually hidden when using vector loads and stores since each load or store requests multiple locations which are pipelined through the network. Furthermore, vector memory references can overlap so that one batch of 128 loads or stores can start before the previous has finished.

The CRAY has multiple memory ports per processor (2 on the J90 and 6 ports on the C90). These ports have different functions. Some are for reading and some are for writing, and only a subset of the ports can be used for gathers and scatters (1 on the J90 and 2 on the C90). This means that the bandwidth for irregular access patterns is not as high as for regular access patterns.

Table 2 shows the $(d, \mathbf{x})$-BSP parameters for the CRAY C90 and CRAY J90. The gap $g$ is measured experimentally and the other values are available from the machine

| | C90 | C90* | J90 |
|---|---|---|---|
| Processors | 16 | 16 | 16 |
| Banks | 1024 | 512 | 1024 |
| Memory | SRAM | SRAM | DRAM |
| Clock period | 4.2 nsec | 4.2 nsec | 10.0 nsec |
| $g$ | 1.2 | 2.5 | 1.8 |
| $d$ | 6 | 6 | 14 |

Table 2: The parameters for the CRAY C90 and CRAY J90. The gap and delay are measured in clock periods. The C90* is the configuration of the C90 available to us at the Pittsburgh Supercomputing Center, which has only half the memory banks, memory ports and network of a full configuration.

specifications. We are interested in the gap for irregular access patterns, in particular ones that require gathers and scatters. As mentioned earlier the regular accesses can be counted in the work term.[4] We base the gap on the time for a scatter to random locations using all the processors, where the destination addresses are loaded from memory. The time for a gather is almost the same (within 10%). Our measured gap is somewhat lower than the theoretical peak performance for gather or scatter operations due to the fact that the memory system is fairly saturated. This saturation effect has also been noted by Bucher and Simmons [10]. Our experiments show that the gap measured for random access patterns reasonably model other irregular patterns.

We have performed several experiments of memory access patterns to quantitatively compare the predictions given by the $(d, \mathbf{x})$-BSP with running times on the CRAY C90 and CRAY J90. The experiments were selected to test various aspects and extremes of the model. Figure 6 summarizes the experiments. For most experiments the measured times closely match the predicted times. For one of the experiments, 4c, the numbers differ by up to a factor of 2.5 due to the effects of the network, which are discussed. In all our synthetic experiments we assume that the work term $W$ can be ignored since on the CRAY it is typically subsumed by the $g \cdot S$ term. In our algorithm experiments described in Section 6, the work term is measured experimentally.

For all experiments we randomize the injection of memory requests within the processors. All the experiments are based on using the scatter operation, although experiments with the gather operation have given almost identical results. The patterns we are interested in cannot be created with strided access—timings for various strided access patterns can be found elsewhere [15, 48]. All experiments were run on a dedicated 8 processor

[4] In our experiments we only use unit stride regular loads and stores.

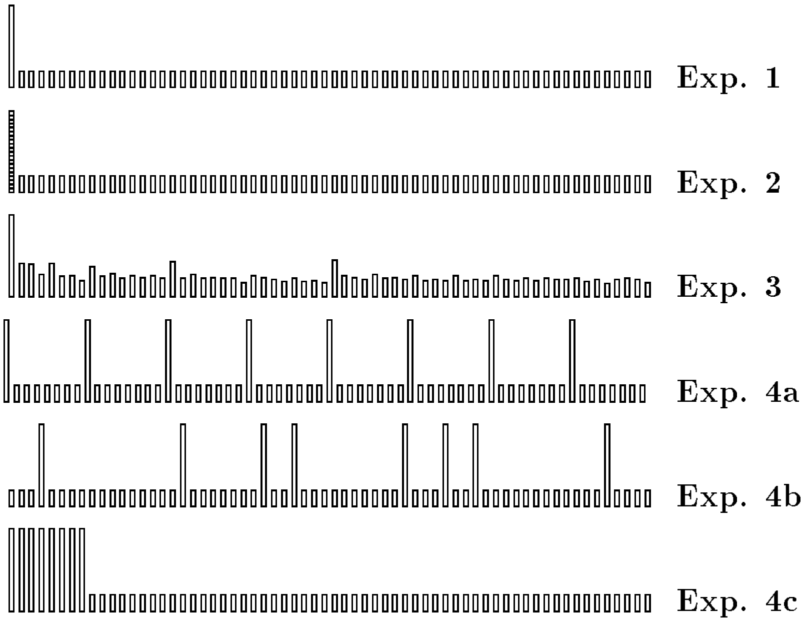
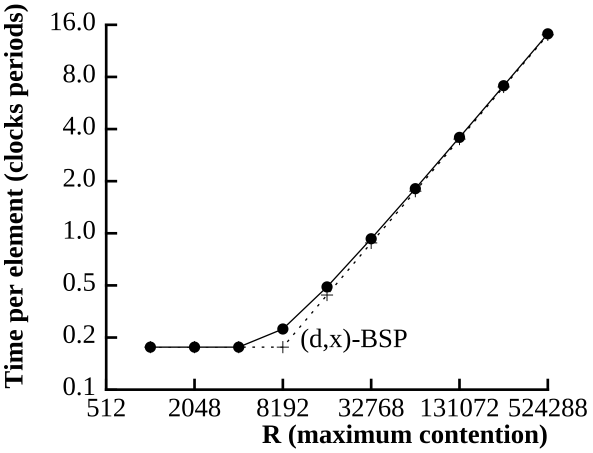



Figure 6: Summary of the experiments. Each bar represents the load on one memory bank. A shaded bar (leftmost bar in Exp. 2) represents multiple different locations being written to the bank while a clear bar (all others) represents a single location being written.

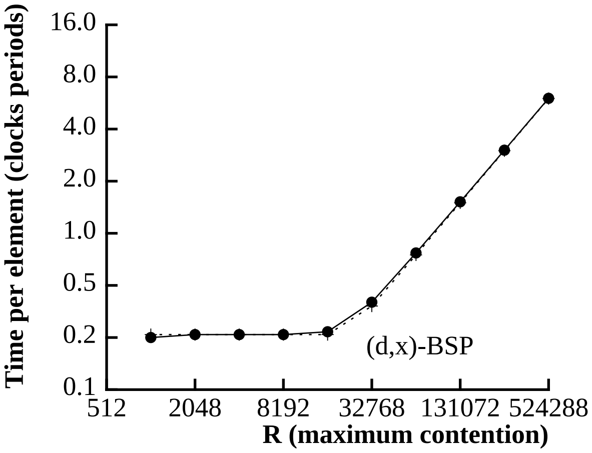
system. All graphs are for the CRAY J90 except where noted—CRAY C90 results are qualitatively similar. For all experiments $S = 64K$ and the periodicity parameter $L$ is negligible.

**Experiment 1:** The first of the experiments is used to verify the $(d, \mathbf{x})$-BSP time equation $T = \max(g \cdot S, d \cdot R)$ over a range of $R$. The experiment consisted of writing one location with load $R$; the remaining work is spread across $B - 1$ memory locations, one memory location per remaining bank. $R$ is varied and $S$ is kept constant. For this experiment, the model is accurate over a range of contentions, as shown in Figure 7. However, at the knee of the curve, the measurements for the CRAY J90 are slightly higher than predictions due to small additive effects of the memory and processor terms.

**Experiment 2:** The second experiment is used to verify that the time is determined by the maximum contention at a bank, independent of whether all the contention is to one location within the bank or is to many locations. The experiment consisted of sending a single request to $R$ different locations within a single bank; the remaining work is spread across $B - 1$ memory locations, one memory location per remaining bank. Again, $R$ is varied and $S$ is kept constant. This differs from the previous experiment only in




(a)




(b)

Figure 7: Experiment 1: Measured and predicted times on (a) the CRAY J90 and (b) the CRAY C90 over a range of contentions (log-log scale). The measured time (shown with a solid curve) is very close to the maximum of the time spent at the processors and memory. The knee in the curve is where the dominant term switches.

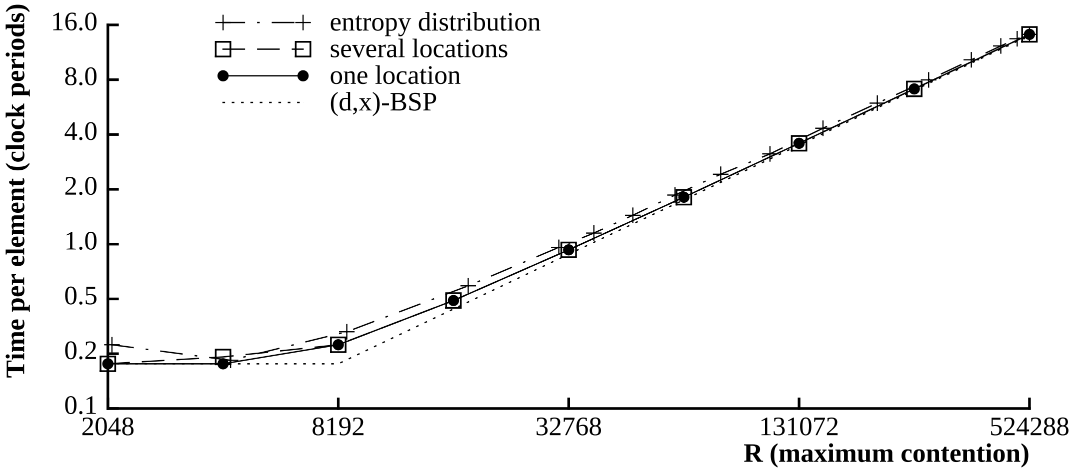



Figure 8: Comparison on the CRAY J90 of experiment 1, experiment 2, and experiment 3: one measuring the time where each bank contains at most one active memory location, one where the accesses to each bank are spread over many memory locations, and one using successive ANDings of random keys. As expected, the curves are nearly identical.

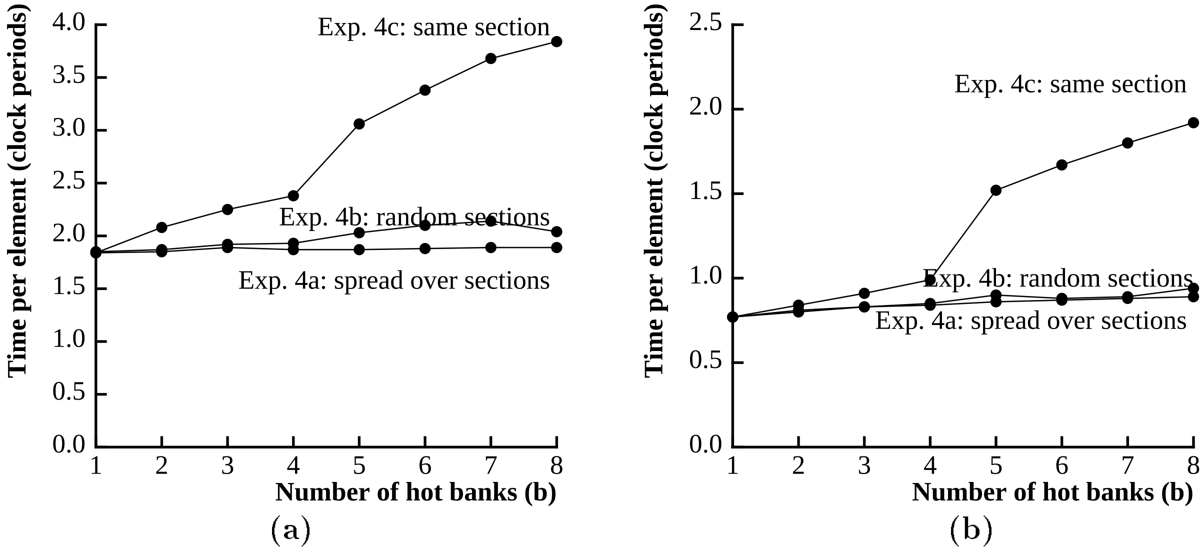



Figure 9: Time on (a) the CRAY J90 and (b) the CRAY C90 as a function of the number of hot banks when $R$ is constant ($R = S \cdot p/8$) The three curves are for different distributions of hot banks across the network and show the effect of network contention.

that the load on the "hot" bank is due to multiple memory locations rather than multiple elements to the same location. The results shown in Figure 8 verify that the performance is not affected by whether the $R$ term is dominated by location contention or module map contention.

**Experiment 3:** To verify that the running time can be accurately predicted for less regular distributions of memory accesses, we constructed an experiment using the entropy distributions suggested by Thearling and Smith [50]. The distributions are generated by starting with a set of random keys and then bitwise ANDing together each key with another key selected at random. Iterating this process generates a family of distributions each with a higher contention than the previous and eventually all keys become 0. The experiment was run over the whole family. Figure 8 shows the time as a function of this maximum contention. The predictions are slightly less accurate than those in Experiments 1 and 2, but are still within 30% of the measurements.

**Experiment 4:** To verify that the running time is determined by the maximum contention $R$, independent of the average contention and distribution of contention, we send $R$ requests to one location within each of $b$ banks and send an equal portion of the remaining work to $B-b$ locations all in different banks. $R$ and $S$ are kept constant, while $b$ is varied. We tried three versions of this experiments differing in how the high-contention banks are distributed across the network: (a) evenly distributed across the network, (b) randomly distributed across the banks, and (c) all within the same section of the network.

Figure 9 shows the results of the experiment using the worst-case value for $R$. Versions (a) and (b) are quite close to the predicted performance. Version (c), however, is up to a factor of 2.5 off from the prediction because of congestion at one of the *subsections* of the network. A more refined model would be needed to take account of this [46, 47], but the experiment shows that even in what we expect to be the worst case the predictions are not catastrophic. Note that when memory is mapped at random into banks, an issue that is discussed in the next section, the situation described in version (c) is unlikely.

# 4 Using random memory mappings

Randomly mapping memory locations to memory banks is a standard technique to reduce module map contention (contention due to multiple memory locations being mapped to the same bank) in simulations of shared memory on machines with a fixed set of memory modules (see, *e.g.*, [36, 32, 41, 52, 2, 40, 42, 35, 28, 24]). The primary advantage of random mapping is that it ensures that concurrently requested memory locations will likely be distributed evenly across the banks. In this section we study to what extent we can ignore the module map contention ($\mu$) when randomly mapping memory to banks. In particular we consider the ratio of the time including module map contention to the time excluding it. We call this ratio the *map contention ratio* ($c$), and in the $(d, \mathbf{x})$-BSP it can be expressed as

$$c \;=\; \frac{\max(W,\ g \cdot S,\ d \cdot R,\ L)}{\max(W,\ g \cdot S,\ d \cdot k,\ L)} \tag{1}$$

We are interested in how bad this ratio is for various machine parameters ($p$, $g$, $d$, $\mathbf{x}$) and memory access patterns. We show that for the CRAY, which has a reasonably high expansion factor, $c$ is small. The results in this section are generalized in the context of the QRQW PRAM simulation in the next section.

To derive an equation that bounds the map contention ratio we consider the memory access pattern in which all memory locations that are being accessed have the same contention. This pattern seems to maximize the map contention ratio for a given $k$ (maximum contention at any memory location). We call this a *uniform distribution* of requests and assuming each processor is making $S$ requests, then a total of $m = Sp/k$ locations are being accessed. We are interested in the map contention ratio as a function of $m$—the worst ratio over $m$ will give us the worst overall ratio. If we assume the cost is dominated by the send and contention terms we can simplify equation 1 to

$$c(m) = \frac{\max(mz, \mu)}{\max(mz, 1)}$$

where $z = g/dp$; $z$ is a constant of the machine. The module map contention $\mu$ can be expressed in terms of the function $Urn(m,n)$, which is the expected maximum sized bucket when throwing $m$ balls into $n$ buckets at random. More specifically, let $m$ balls be thrown independently at random into $n$ urns, and let $y$ be the random variable representing the maximum number of balls in any one urn; then, $Urn(m,n) = \mathbf{E}\,(y)$, the expected value of $y$. We can estimate:

$$c(m) \approx \frac{\max(mz, Urn(m,B))}{\max(mz,1)} \tag{2}$$

Figure 10 shows both the predicted and measured values of $c$ as a function of $m$ for the CRAY C90 and the CRAY J90. The measured ratio is based on the average over 20 trials. The results show that for all $m$, $c(m)$ is at most 2.0 on the C90 and 2.5 on the J90, and for most patterns it is close to 1. This argues that for many practical purposes we can ignore the module map contention when using random mappings on the CRAY—inaccuracies in predictions from other sources are likely to dominate. Intuitively, the reason for the peak in the graph is that as we increase the number of "hot" locations ($m$) past the peak, the load at each location decreases and eventually the first term in the equation $\max(g \cdot S,\ d \cdot k)$ dominates. As we decrease $m$ below the peak, it becomes less likely that multiple locations will be mapped to the same bank. In particular when $m < \sqrt{B}$ it becomes less likely that more than one location will be mapped to a single bank. The slight difference between the predicted and measured times is due to effects in the network as discussed in the previous section. In particular, at these small $m$ it is reasonably likely that the locations are not only imbalanced across the banks, but are also imbalanced across the *sections* in the network, causing backup at the source due to section conflicts.

What is more interesting is to see how the worst case $c$ is affected by the machine parameters. Equation 2 is maximized when $mz = 1$ (*i.e.*, $m = dp/g$), giving

$$c_{\max} = Urn\left(\frac{1}{z}, B\right) = Urn\left(\frac{dp}{g}, p\mathbf{x}\right) \quad .$$

Figure 2 shows $c_{\max}$ as a function of $\mathbf{x}$ for $d$ and $g$ set to the parameters of the CRAY C90 and CRAY J90 and $p$ set to 16. As can be seen, it is helpful to have an expansion factor beyond the $(d,\mathbf{x})$-balanced ratio ($\mathbf{x} = d/g$) in order to minimize the impact on the running time of module map contention.

**Pseudo-random memory mapping**. As with previous work, we assumed above that the memory locations are hashed to memory banks using a truly random mapping. In practice, however, the mapping cannot be truly random, since it should be efficiently computable for every memory address.

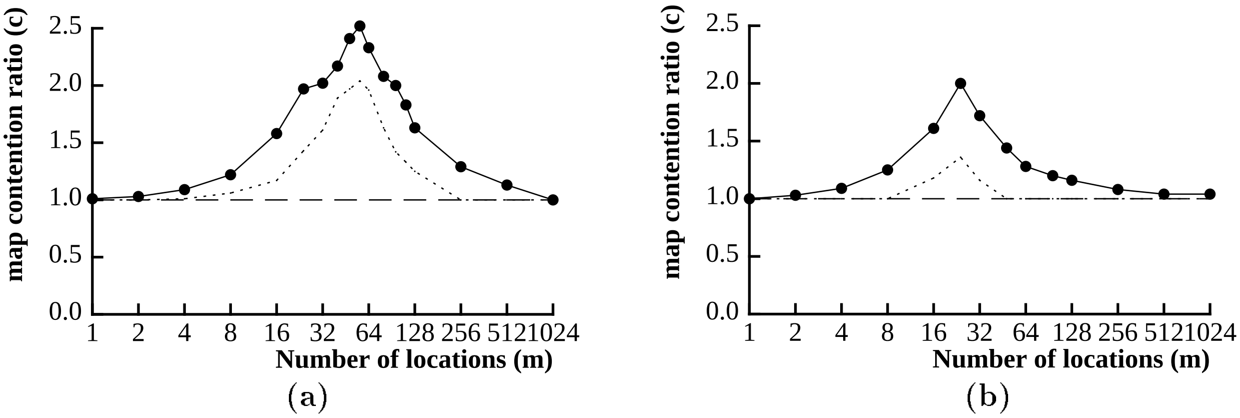



Figure 10: The measured and predicted map contention ratio $c$ for (a) J90 and (b) C90. The measured ratio is taken as the ratio of the measured running time on the CRAY to the equation $\max(g \cdot S,\ d \cdot k)$, which ignores module map contention (solid curves). The predicted is given by Equation 2. This is for a uniform distribution to $m$ locations using random mappings from locations to banks. $S$ is kept constant; $p = 8$.

The TERA design is planned to provide hardware support for hash functions to be used for pseudo-random mapping of memory locations to memory banks; the Fujitsu uVP on the Meiko node already has optional hardware hashing. The CRAY does not supply hardware to perform the pseudo-random mapping of the memory locations to banks. For some algorithms, however, it is possible to get the same effect by randomly permuting the input and some of the intermediate results. In others, the nature of the algorithm results in random mapping without any additional steps (see examples in Section 6). For other algorithms, computing a pseudo-random hash in software is not prohibitive.

The actual evaluation costs for a variety of hash functions on the CRAY C90 are given in Table 3. When hash functions are used for pseudo-random mapping of memory locations to memory banks, it is important that they exhibit favorable properties for any given input (*i.e.* that they are "universal"). The function $h_a^1$, which is called the *multiplicative hashing scheme* in [34, p. 509], was recently shown by Dietzfelbinger *et al.* [20] to be 2-universal in the sense of Carter and Wegman [12]: for any two distinct numbers $x, y \in [0..2^u - 1]$, $\textbf{Prob}\,(h_a^1(x) = h_a^1(y)) \leq 1/2^{m-1}$; *i.e.*, the collision probability is approximately the same as for a random mapping. The actual choice of a hash function may be influenced by several factors, including its degree of universality, its evaluation cost, and its congestion behavior, both theoretically (see [19]) and experimentally (see [21]).

| Hash Function | | $T/n$ |
|---|---|---|
| Linear | | |
| $h^1_{a,b}(x)$ | $= ((ax+b) \bmod 2^u) \text{ div } 2^{u-m}$ | 1.8 |
| $h^1_a(x)$ | $= (ax \bmod 2^u) \text{ div } 2^{u-m}$ | 1.8 |
| Quadratic | | |
| $h^2_{a,b,c}(x)$ | $= ((ax^2+bx+c) \bmod 2^u) \text{ div } 2^{u-m}$ | 3.4 |
| $h^2_{a,b}(x)$ | $= ((ax^2+bx) \bmod 2^u) \text{ div } 2^{u-m}$ | 3.4 |
| Cubic | | |
| $h^3_{a,b,c,d}(x)$ | $= ((ax^3+bx^2+cx+d) \bmod 2^u) \text{ div } 2^{u-m}$ | 6.7 |
| $h^3_{a,b,c}(x)$ | $= ((ax^3+bx^2+cx) \bmod 2^u) \text{ div } 2^{u-m}$ | 6.7 |

Table 3: The evaluation cost of software implementations of various hash functions in terms of clock cycles per element (for each CRAY C90 processor). The functions map items $x$ from the domain $[0..2^u]$ into the range $[0..2^m]$, and $a$, $b$, $c$, and $d$ are odd numbers selected at random from $[1..2^u - 1]$.

# 5 High-level programming model

In this section and the next, we explore scenarios under which a high-level model for algorithm design, the QRQW PRAM, can be effectively mapped onto a $(d, \mathbf{x})$-BSP and hence onto high-bandwidth machines.

The QRQW PRAM [24] is a variant of the well-studied PRAM model (see *e.g.* [33, 31]) that allows for concurrent reading and writing to shared memory locations, but assumes that multiple reads/writes to a location queue up and are serviced one at a time (named the "queue-read queue-write (QRQW)" contention rule in [24]). Specifically, the QRQW PRAM consists of $p$ processors communicating by reading and writing words from a shared memory. Processors also have local memory for use with local operations. A QRQW PRAM computation consists of a series of supersteps separated by barrier synchronizations. In each superstep the processors can perform local computations and make a set of pipelined, global memory requests. Requests made in one superstep will complete prior to the start of the next superstep. The time charged for a superstep is calculated as follows. Let $W$ be the maximum amount of local work performed by any one processor in the superstep, let $S$ be the maximum number of global memory requests made by any one processor, and let $k$ be the maximum number of requests to any one location. Then the time for the superstep is $\max(W, S, k)$.

The QRQW PRAM is an even simpler model than the $(d, \mathbf{x})$-BSP. Unlike the BSP or $(d, \mathbf{x})$-BSP models, the QRQW PRAM memory is not explicitly partitioned into memory banks—each processor has equal access to each memory location. Furthermore the

QRQW PRAM has no $g$ or $L$ parameters. The simulation of the QRQW PRAM on the $(d, \mathbf{x})$-BSP hides the latency $L$ by using a factor of at least $L$ more "virtual processors" on the QRQW PRAM than are available on the $(d, \mathbf{x})$-BSP. The QRQW PRAM is more powerful than the well-studied EREW PRAM (which requires $k = 1$ at each step) but less powerful than the well-studied CRCW PRAM (which permits arbitrary $k$ without charge). It was argued in [24] that the QRQW contention rule more accurately reflects the contention capabilities of most machines than the EREW or CRCW contention rules.

The interesting question is under what conditions can one use the simpler QRQW PRAM instead of the $(d, \mathbf{x})$-BSP for modeling algorithms. In this section we consider theoretical results on when the $(d, \mathbf{x})$-BSP can effectively emulate the QRQW. Then in Section 6, we discuss experimental results regarding the implementation of two QRQW PRAM algorithms from [23] on the CRAY. These experimental results complement the general emulation results, by demonstrating two of the scenarios under which the three metrics (*i.e.* $W, S, k$) of the QRQW PRAM model are sufficient to accurately predict performance on the CRAY.

**Overview of the emulation results.** Recall that a machine is $(d, \mathbf{x})$-balanced if $\mathbf{x} = d/g$, *i.e.* the total bandwidth available at the processors and network matches the total bandwidth available at the memory banks. Let $d_g$ be the bank delay normalized to the gap parameter, *i.e.* $d_g = d/g$. We present in this section two emulations of the QRQW PRAM on the $(d, \mathbf{x})$-BSP, one for the case where $\mathbf{x} \geq d_g$ and one for the case where $\mathbf{x} < d_g$. In the former case, we observe that any step-by-step emulation must incur an overhead of $g$ in the work performed, and we provide an emulation of the QRQW PRAM on the $(d, \mathbf{x})$-BSP that matches this work overhead. Thus when $g$ is a small constant, as when modeling high-bandwidth machines, the emulation is work-optimal. The slowdown in the emulation is a nonlinear function of the parameters of the $(d, \mathbf{x})$-BSP; the slowdown is minimized when $\mathbf{x} \geq d_g \cdot 2^{d_g}$, in which case $d_g$ is only an additive term in the slowdown.

As for the case when $\mathbf{x} < d_g$, we observe that any emulation must also incur overhead due to the insufficient bandwidth at the memory banks, and we provide an emulation whose work bounds match the lower bound that we prove.

All of our emulation results (upper and lower bounds) apply as well to the EREW PRAM. These results extend the previous results in [52] and [24] that showed that when $g$ is a small constant, there is a work-optimal emulation of the EREW PRAM and QRQW PRAM, respectively, on the original BSP model in which the slowdown in the emulation is $\Theta(\lg p + L)$.

## 5.1 Work-optimal QRQW PRAM emulation on $(d, \mathbf{x})$-BSP

The following theorem presents an emulation of the QRQW PRAM on a $(d, \mathbf{x})$-BSP for the case when $\mathbf{x} \geq d/g$, where $g$ is the gap parameter for the $(d, \mathbf{x})$-BSP. When $g$ is a small constant, the emulation is work-preserving (*i.e.* the work performed on the $(d, \mathbf{x})$-BSP is within constant factors of the work performed on the QRQW PRAM).

**Theorem 5.1 (work-optimal QRQW simulation)** *Consider a $p$-processor $(d, \mathbf{x})$-BSP with gap parameter $g$ and periodicity factor $L$, such that $d_g \leq \mathbf{x} \leq p^{\bar{c}}$, for some constant $\bar{c} > 0$, where $d_g = d/g$. Let*

$$\delta = \begin{cases} d_g & \text{if} \quad d_g \leq \mathbf{x} \leq 2d_g \\ d_g / \lg(\mathbf{x}/d_g) & \text{if} \quad 2d_g \leq \mathbf{x} \leq d_g 2^{d_g} \\ 1 & \text{if} \quad \mathbf{x} \geq d_g 2^{d_g} \end{cases}$$

*Then for all $p' \geq (\delta \lg p + d_g + L)p$, each step of a $p'$-processor* QRQW PRAM *algorithm running in time $t$ can be emulated on the $p$-processor $(d, \mathbf{x})$-BSP in $O(g \cdot (p'/p) \cdot t)$ time w.h.p.*

*Proof.* The shared memory of the QRQW PRAM is randomly hashed onto the $B = \mathbf{x} \cdot p$ memory banks of the $(d, \mathbf{x})$-BSP. In the emulation algorithm, each $(d, \mathbf{x})$-BSP processor executes the operations of $p'/p$ QRQW PRAM processors.

We first assume that $2d_g \leq \mathbf{x} \leq d_g 2^{d_g}$, and therefore $\delta = d_g / \lg(\mathbf{x}/d_g)$.

Consider the $i$th step of the QRQW PRAM algorithm, with time cost $t_i$. Let $c > 0$ be some arbitrary constant, and let $\alpha = \max\{c + \bar{c} + 1, e\}$. We will show that this step can be emulated on the $(d, \mathbf{x})$-BSP in time at most $\alpha g (p'/p) t_i$ with probability $1 - p^{-c}$.

By the definition of the QRQW PRAM cost metric, we have that both the maximum memory request contention $k$ and the maximum memory request load $S$ are at most $t_i$. For the sake of simplicity in the analysis, we add dummy memory requests to each processor as needed so that it sends exactly $t_i$ memory requests this step. The dummy requests for a processor are to dummy memory locations, with processor $\ell$ sending all its dummy requests to dummy location $\ell$. In this way, the maximum memory request contention $k$ remains at most $t_i$, and the total number of requests is $Z = p' t_i$.

Let $i_1, i_2, \ldots, i_m$ be the different memory locations accessed in this step (including dummy locations), and let $k_j$ be the number of accesses to location $i_j$, $1 \leq j \leq m$. Note that $\sum_{j=1}^{m} k_j = Z$. Consider a memory bank $\beta$. For $j = 1, \ldots, m$, let $x_j$ be an indicator binary random variable which is 1 if memory location $i_j$ is mapped onto the memory bank $\beta$, and is 0 otherwise. Thus, $\mathbf{Prob}\,(x_j = 1) = 1/B$. Let $a_j = k_j / t_i$; $a_j$ is the normalized contention to location $j$. Since $k \leq t_i$, we have that $a_j \in (0,1]$. Let $\Psi_\beta = \sum_{j=1}^{m} a_j x_j$; $\Psi_\beta$, the normalized module load contention to bank $\beta$, is the weighted sum of Bernoulli trials. The expected value of $\Psi_\beta$ is

$$\mathbf{E}\left(\Psi_\beta\right) = \sum_{j=1}^{m} \frac{a_j}{B} = \frac{1}{\mathbf{x}p} \sum_{j=1}^{m} \frac{k_j}{t_i} = \frac{1}{\mathbf{x}p} \cdot \frac{Z}{t_i} = \frac{p'\,t_i}{\mathbf{x}\,p\,t_i} = \frac{p'}{\mathbf{x}p}.$$

To show that it is highly unlikely that the module load contention of bank $\beta$ greatly exceeds this expected value, we will use the following theorem by Raghavan and Spencer, which provides a tail inequality for the weighted sum of Bernoulli trials:

**Theorem 5.2 ([39])** *Let $a_1, \ldots, a_m$ be reals in $(0,1]$. Let $x_1, \ldots, x_m$ be independent Bernoulli trials with $\mathbf{E}\left(x_j\right) = \rho_j$. Let $\Psi_\beta = \sum_{j=1}^{m} a_j x_j$. If $\mathbf{E}\left(\Psi_\beta\right) > 0$, then for any $\nu > 0$*

$$\mathbf{Prob}\left(\Psi_\beta > (1+\nu)\mathbf{E}\left(\Psi_\beta\right)\right) < \left(\frac{e^\nu}{(1+\nu)^{(1+\nu)}}\right)^{\mathbf{E}\left(\Psi_\beta\right)} . \tag{3}$$

We apply Theorem 5.2 with $\rho_j = 1/B$, and set

$$\nu = \alpha \frac{\mathbf{x}}{d_g} - 1 \ ,$$

implying

$$(1+\nu)\mathbf{E}\left(\Psi_\beta\right) = \alpha \frac{\mathbf{x}}{d_g} \cdot \frac{p'}{\mathbf{x}p} = \frac{\alpha p'}{d_g p} \ . \tag{4}$$

Therefore,

$$\begin{aligned}
\mathbf{Prob}\left(\Psi_\beta > \frac{\alpha p'}{d_g p}\right) \quad &\overset{(3),(4)}{<} \quad \left(\frac{e}{(1+\nu)}\right)^{(1+\nu)\mathbf{E}\left(\Psi_\beta\right)} \overset{(4)}{=} \left(\frac{\alpha \mathbf{x}}{e d_g}\right)^{-\frac{\alpha p'}{d_g p}} \\
&\overset{\alpha \geq e}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha p'}{d_g p}} \overset{\mathbf{x} > d_g}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha}{d_g}(\delta \lg p + d_g + L)} \\
&\overset{\mathbf{x} > d_g}{\leq} \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha}{d_g}\delta \lg p} = \left(\frac{\mathbf{x}}{d_g}\right)^{\frac{-\alpha}{\lg(\mathbf{x}/d_g)} \lg p} = p^{-\alpha} \\
&\leq \quad p^{-(c+\bar{c}+1)} = \frac{p^{-(c+1)}}{p^{\bar{c}}} \\
&\overset{\mathbf{x} \leq p^{\bar{c}}}{\leq} \frac{p^{-(c+1)}}{\mathbf{x}} \ .
\end{aligned}$$

Note that $R_\beta$, the module load contention to bank $\beta$, is

$$R_\beta = \sum_{j=1}^{m} x_j k_j = \Psi_\beta \cdot t_i \ .$$

Therefore,

$$\mathbf{Prob}\left(R_\beta > \frac{\alpha\, p'\, t_i}{d_g\, p}\right) < \frac{p^{-(c+1)}}{\mathbf{x}} \ ,$$

and hence

$$\mathbf{Prob}\left(R > \frac{\alpha\, p'\, t_i}{d_g\, p}\right) \ \leq \ B \cdot \mathbf{Prob}\left(R_\beta > \frac{\alpha\, p'\, t_i}{d_g\, p}\right) < B \cdot \frac{p^{-(c+1)}}{\mathbf{x}} = p^{-c} \ .$$

The time of the $(d, \mathbf{x})$-BSP step to emulate a QRQW step is $T = \max(W,\ g \cdot S,\ d \cdot R,\ L)$. Thus for step $i$, $T_i = \max((p'/p)t_i,\ g(p'/p)t_i,\ dR,\ L)$. Since $p'/p > L$, it follows from the above that

$$\mathbf{Prob}\left(T_i \leq \alpha\, g\, (p'/p)\, t_i\right) \geq 1 - p^{-c} \ .$$

We next consider the case where $d_g \leq \mathbf{x} \leq 2d_g$, and therefore $\delta = d_g$. In this case we take $\alpha = \max\{c + \bar{c} + 1, 2e\}$, and the proof proceeds as above except that we make use of the fact that

$$\left(\frac{\alpha \mathbf{x}}{e d_g}\right)^{-\frac{\alpha p'}{d_g p}} \leq 2^{-\frac{\alpha p'}{d_g p}} \leq 2^{-\frac{\alpha}{\delta}(\delta \lg p + d_g + L)} \leq 2^{-\alpha \lg p} = p^{-\alpha} \ .$$

It remains to consider the case where $\mathbf{x} \geq d_g 2^{d_g}$. Consider a partition of the memory banks into $\mathbf{x}' = d_g 2^{d_g}$ sets, each denoted as a *memory super-bank*. The indicator random variables $x_j$ and the module load contention are defined with respect to the memory super-banks analogously to their original definitions; denote the latter as $R'$. The above analysis for $R$ clearly holds for $R'$. Since $R \leq R'$, the theorem follows. ■

The following observation shows that the overhead of $g$ in the above emulation is unavoidable, even for the EREW PRAM.

**Observation 1** *Let $p' \geq p$. Any simulation of one step of a $p'$-processor* EREW *or* QRQW PRAM *with time cost $t$ on a $p$-processor* $(d, \mathbf{x})$-BSP *requires* $\max\{g(p'/p)t, L\}$ *time in the worst case.*

*Proof.* Consider a step in which each of the $p'$ QRQW processors performs $t$ memory requests to distinct locations. The time on the $(d,\mathbf{x})$-BSP is at least $\max\{g(p'/p)t, L\}$. The EREW proof is the same, taking $t = 1$. ∎

## 5.2 QRQW PRAM emulation with small x

We next consider the case where the bandwidth at the memory banks is less than the bandwidth at the processors and network, *i.e.* $\mathbf{x} < d_g$. We present an emulation whose work bound is within a constant factor of the best possible.

**Theorem 5.3 (QRQW simulation with small x)** *Consider a p-processor* $(d,\mathbf{x})$-BSP *with gap parameter g and periodicity factor L, such that* $1 \leq \mathbf{x} < \min\{d_g, p^{\bar{c}}\}$, *for some constant* $\bar{c} > 0$, *where* $d_g = d/g$. *Then for all* $p' \geq (\mathbf{x}\lg p + d_g + L)p$, *each step of a* $p'$-*processor* QRQW PRAM *algorithm running in time t can be emulated on the p-processor* $(d,\mathbf{x})$-BSP *in* $O(\max\{g, d/\mathbf{x}\} \cdot (p'/p) \cdot t)$ *time w.h.p.*

*Proof.* The theorem can be proved following the lines of the theorem given in [24] for emulating the QRQW PRAM on the (standard) BSP, by generalizing the argument in [24] to handle the $d$ and $\mathbf{x}$ parameters, and making the $g$ parameter explicit in the analysis. Instead, to unify the proofs of Theorem 5.1 and Theorem 5.3, we prove the latter theorem using an analysis similar to the proof of the former theorem, as follows.

As in the proof of Theorem 5.1, the shared memory of the QRQW PRAM is randomly hashed onto the $B = \mathbf{x} \cdot p$ memory banks of the $(d,\mathbf{x})$-BSP. In the emulation algorithm, each $(d,\mathbf{x})$-BSP processor executes the operations of $p'/p$ QRQW PRAM processors.

Consider the $i$th step of the QRQW PRAM algorithm, with time cost $t_i$. Let $c > 0$ be some arbitrary constant, and let $\alpha = \max\{c + \bar{c} + 1, 2e\}$. We will show that this step can be emulated on the $(d,\mathbf{x})$-BSP in time at most $\max\{g(p'/p)t_i,\ \alpha(d/\mathbf{x})(p'/p)t_i\}$ with probability $1 - p^{-c}$.

The proof proceeds exactly as in the proof of Theorem 5.1: we add dummy requests as needed, define indicator binary random variables $x_j$ for each memory bank $j$, define $\Psi_\beta$, and show that $\mathbf{E}(\Psi_\beta) = p'/(\mathbf{x}p)$. We apply the Raghavan and Spencer theorem, but with $\nu = \alpha - 1$. This yields

$$\begin{aligned}\mathbf{Prob}\left(\Psi_\beta > \frac{\alpha p'}{\mathbf{x}p}\right) &< \left(\frac{\alpha}{e}\right)^{-\frac{\alpha p'}{\mathbf{x}p}} \overset{\alpha \geq 2e}{\leq} 2^{-\frac{\alpha}{\mathbf{x}}(\mathbf{x}\lg p + d_g + L)} \\ &\leq p^{-\alpha} \leq p^{-(c+\bar{c}+1)} \overset{\mathbf{x} \leq p^{\bar{c}}}{<} \frac{p^{-(c+1)}}{\mathbf{x}} .\end{aligned}$$

It follows as in the previous proof that

$$\mathbf{Prob}\left(R > \frac{\alpha\, p'\, t_i}{\mathbf{x}\, p}\right) < p^{-c} \ .$$

The time, $T_i$, of the $(d,\mathbf{x})$-BSP step to emulate QRQW step $i$ is $\max((p'/p)t_i,\ g(p'/p)t_i,\ dR,\ L)$. Since $p'/p > L$, we have that

$$\mathbf{Prob}\left(T_i \leq \max\left\{g \cdot \frac{p'}{p} \cdot t_i\ ,\ \alpha \cdot \frac{d\, p'}{\mathbf{x}\ p} \cdot t_i\right\}\right) \geq 1 - p^{-c} \ .$$

The theorem follows. ∎

Note that Observation 1 shows that the overhead of $g$ in the above emulation is unavoidable. The following observation shows that the overhead of $d/\mathbf{x}$ in the above emulation is also unavoidable.

**Observation 2** *Let $p' \geq p$. Any simulation of one step of a $p'$-processor* EREW *or* QRQW PRAM *with time cost $t$ on a $p$-processor* $(d,\mathbf{x})$-BSP *requires $d \cdot \lceil tp'/\mathbf{x}p \rceil$ time in the worst case.*

*Proof.* Consider a step in which each of the $p'$ QRQW processors perform $t$ memory requests, such that all $p't$ requests are to distinct locations in the shared memory. Since there are $m = p't$ locations distributed among $\mathbf{x}p$ memory banks, then regardless of the mapping of locations to banks, there exists at least one bank $j$ such that $\mu_j \geq \lceil m/\mathbf{x}p \rceil$. Therefore, the time on the $(d,\mathbf{x})$-BSP is at least $\mu_j d \geq d \cdot \lceil tp'/\mathbf{x}p \rceil$. The EREW proof is the same, taking $t = 1$. ∎

# 6 Algorithm experiments

In this section, we present results of experiments on four algorithms: QRQW algorithms for binary search and random permutation, and CRCW algorithms for sparse matrix multiplication and finding connected components of a graph. The $(d,\mathbf{x})$-BSP model can be used for accurately predicting the running time of each algorithm, and in the case of the binary search, for determining a good setting for the "fatness" parameter.

The four problems considered — binary searching, graph connectivity, sparse matrix vector multiply, and generating a random permutation — are intended to serve as representative of the most basic unstructured problems. Such core problems arise in a number of unstructured applications, and are often the bottlenecks in such applications. For example, the sparse matrix vector product is the dominant cost in the Conjugate Gradient (CG) methods on unstructured meshes. In fact our code on the Cray vector multiprocessors is the fastest reported code for the NAS CG benchmark [44]. As another example, the graph connectivity problem is the dominant cost in simulating Ising Spin models using the Swendsen Wang algorithm [49].

The four problems arise from diverse domains, with the intention that the memory access patterns of the algorithms studied will reflect patterns exhibited by a large class of unstructured algorithms. Further experimentation is required to validate the extent to which these four problems are indeed representative of a larger class of unstructured algorithms.

**Binary search:** The first QRQW algorithm is a simple parallel binary search to look up $n$ keys in a balanced binary search tree of size $m$ [23]. Such binary searching is an important substep in several algorithms for sorting and merging (*e.g.* [43]). The algorithm replicates nodes of the search tree to avoid contention, and at each level selects one of the replicated nodes at random. This is an interesting problem from the point of view of the QRQW PRAM and the $(d, \mathbf{x})$-BSP since the amount of replication needed will depend on the contribution of the contention term to the running time, and in general will present a tradeoff. On the CRCW PRAM there is no need for replication.

To design an optimized algorithm that uses binary searching we would like to use the $(d, \mathbf{x})$-BSP model to predict the running time of the binary search for different amounts of replication. In the experiment, the root is replicated $f$ times (the "fatness"), and each level below the root is replicated half as many times as the level above. Thus there are $\max(2^i, f)$ nodes at level $i$ of the fattened tree. We consider for simplicity the case where the number of nodes at each level is less than the number of banks. Assuming an equal number of lookups to each key, the expected time per lookup is:

$$\sum_{i=0}^{\lceil \lg m \rceil - 1} \max\left(\frac{cg \cdot S}{n}, \frac{d \cdot \mathbf{E}(R)}{n}\right) = \sum_{i=0}^{\lceil \lg m \rceil - 1} \max\left(\frac{cg}{p}, \frac{d}{\max(2^i, f)}\right) \quad ,$$

where $c$ is approximately 3.9 on the CRAY J90 and 1.4 on the C90*. Figure 11 shows that the predictions are very accurate.

**Random permutation:** The second QRQW algorithm generates random permutations using a "dart throwing" algorithm. The algorithm first generates $n$ random indices in the range $[0..cn]$ for some constant $c$ (2 in our experiments). Each element $i$ then writes its self-index into a destination array at the location specified by the $i$th random-index. Elements for which there are no collisions are considered done and drop out. Elements

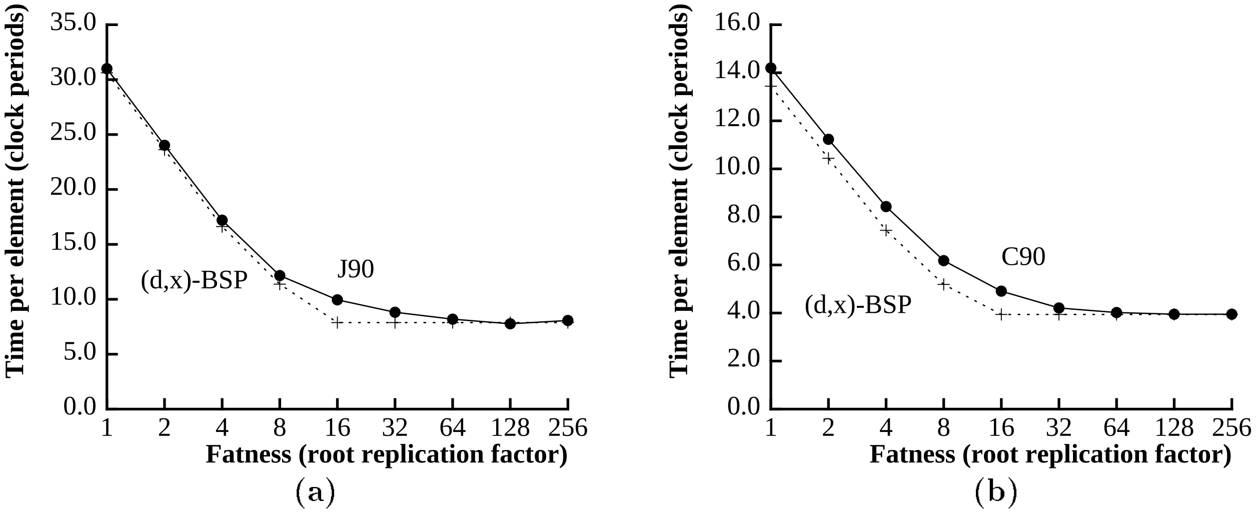



Figure 11: Predicted and measured times for a binary search fat-tree algorithms, as a function of the "fatness" of the tree for a table of size 256. The solid curve shows measured times on 8 processors of the (a) CRAY J90 and (b) CRAY C90 and the dotted curve shows predicted times.

for which there are collisions repeat another round. The rounds continue until there are no elements left. At this point the values written into the destination are packed into contiguous locations, producing the index for the random permutation. The algorithm runs in $O(n/p + \lg n)$ time on a QRQW PRAM [23].

In our experiments we compare the running time of the algorithm to an algorithm designed for the EREW model (a sorting-based algorithm, which is the most practical EREW algorithm in the literature, to the best of our knowledge). The EREW algorithm is based on a radix sort that ranks keys with $2.5 \cdot \lg n$ bits and checks for duplicate keys. This experiment illustrates that by allowing a controlled amount of contention to memory locations, we can get a faster algorithm than in a scenario where we avoid such contention altogether. (A similar experiment, on the MasPar MP-1, was reported in [23]; for the EREW algorithm, the system sort was used.) Results of the experiments are shown in Figure 12. (The timings do not include the time for random number generation; however, the two algorithms require approximately the same number of random bits.)

Using a performance model for radix-sort adapted from [53], the predicted running time for the EREW algorithm is given by:

$$t_{EREW} \quad = \quad 400000 + \left\lceil \frac{2.5 \cdot \lg n}{\lg(n/p) - 8} \right\rceil \cdot (3.5 + 7 \cdot g) \cdot n/p \qquad (5)$$

where the 2.5 comes from the number of bits, 8 comes from latency hiding ($\lg(128) + 1$), 3.5 comes from arithmetic on buckets, and 7 comes from indirect operations for histogram and permutation. The predicted running time for the QRQW dart throwing algorithm

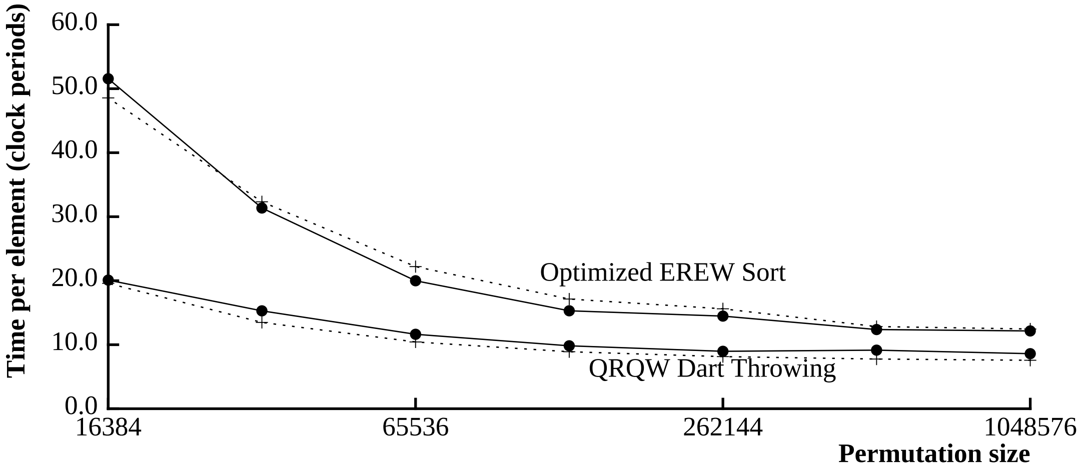



Figure 12: Predicted and measured times on an 8 processor CRAY J90 comparing two algorithms for generating a random permutation: a QRQW dart throwing algorithm and an EREW sorting-based algorithm. The QRQW algorithm performs better over a wider range of problem sizes, and even a simple C implementation outperforms the EREW version, which is based on a highly-optimized radix sort [54]. (The radix sort is currently the fastest implementation of the NAS sorting benchmark [44].) The predicted times are given by equations 5 and 6.

can be derived from experimentally measured arithmetic computation costs, the number of global references (using gather or scatter) and the expected total number of darts thrown. The time is linear in the problem size and is given by:

$$\begin{aligned} t_{QRQW} &= 200000 + (18.7 + 4 \cdot g + w \cdot (5 \cdot g + 12.3)) \cdot n/p \\ &= 200000 + (37.8 + 11.8 \cdot g) \cdot n/p \end{aligned} \tag{6}$$

where $w = 1.55$ is the ratio of total darts thrown to the permutation size (there is less than 1% variation in $w$ as $n$ is varied.)

**Sparse matrix multiplication:** In our third algorithm experiment, we measured the time to multiply a sparse matrix by a dense vector. Our implementation uses a compressed row format containing the number of non-zero elements in each row, and the values of each non-zero matrix element along with its column index. The computation is vectorized using "segmented scan" operations [7], a technique that allows the latency to be hidden regardless of the structure of the matrix. For the purposes of analyzing contention, the most important characteristic of our implementation is that elements from the input vector are gathered based on the column indices of the nonzero matrix element. Elements of the input vector are typically read multiple times. Thus, our formulation of sparse matrix multiplication can be viewed as a CRCW algorithm or as a QRQW algorithm where the contention is equal to the number of elements in the densest column. A dense column can arise in practice from having a global constraint or bias, such as a circuit

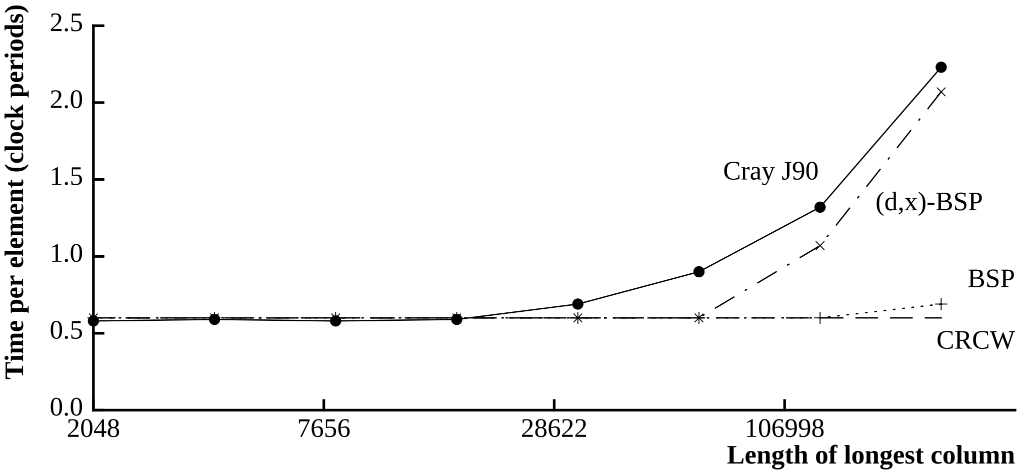



Figure 13: Time per nonzero element for sparse matrix vector multiplication on a matrix with one dense column and an average row size of 7. Measured times are given for an 8 processor CRAY J90, and predicted times are given for the $(d, \mathbf{x})$-BSP, BSP, and CRCW.

with many connections to ground.

In the experiment, we constructed a set of test matrices with one dense column and an average row length of 7. The number of rows and the total amount of work are held constant, while the number of elements in the dense column is varied. Except in the dense column, column indices are selected at random. The predicted running time on the $(d, \mathbf{x})$-BSP is:

$$\max\left(\frac{c}{p}, d \cdot R\right) \quad ,$$

where $c$ is a constant that accounts for the combined effect of the work and the gap. Figure 13 shows measured and predicted times as a function of the length of the dense column, using the measured value $c = 4.8$ for the CRAY J90.

**Connected components:** Our final algorithm experiment measures the contention in Greiner's algorithm for finding the connected components of a graph [26]. The algorithm consists of several phases: hooking nodes together to form a forest, performing repeated shortcutting operations to contract each tree to a single node, contracting the graph to form a new graph that is processed recursively, and expanding the graph to propagate the new labels. Contention can occur in each step of the algorithm and varies considerably depending on characteristics of the input graph and the method used for implementing the contraction and expansion phases. In our experiments, we used a variant of Greiner's suite of test graphs consisting of subsets of 2d and 3d toroidal graphs, random graphs (*i.e.* subsets of complete graphs), and "tertiary" graphs in which each node has three random neighbors. In order to reduce module map contention without hashing the data structures, nodes and edges are randomly permuted and renumbered in a preprocessing step. To simplify the comparison of different phases of the algorithm, we extracted memory access patterns used and timed them on a simple "scatter" operation. Note that as discussed in Section 4, under the circumstances of random memory mapping (in this case, due to the nature of the algorithm) and sufficiently high expansion factor, the module map contention can be ignored without substantial error in the prediction. Figure 1 compares timings on the CRAY J90 to predictions generated using the $(d, \mathbf{x})$-BSP model, but ignoring module map contention. These results have helped to clarify some of the observed performance problems with scaling the CRCW algorithm to a large number of processors. We are currently evaluating a number of heuristics for reducing contention in the CRCW algorithm [53].

# 7 Discussion

This paper studies the effectiveness of using the $(d, \mathbf{x})$-BSP as a simple model for shared memory machines to analyze the memory performance of algorithms. The model accounts for memory bandwidth, memory latency, memory bank delay, and memory bank expansion. We have focused on modeling the effect of these features on unstructured computations, where the memory access patterns are irregular and the lack of locality of memory reference stresses the bandwidth limitations of the shared memory machine. Although the $(d, \mathbf{x})$-BSP abstracts away many machine specific details, our results show that it still gives useful guidance to algorithm designers by providing performance prediction that is reasonably accurate. More accurate predictions can surely be made using more detailed models, but with the tradeoff of complicating analysis and making algorithm design more machine specific and hence less portable.

We verified that the $(d, \mathbf{x})$-BSP reasonably explains the memory performance of the CRAY C90 and J90 on irregular access patterns. Our results can be summarized as follows:

- For low memory contention, high parallel slackness, and random mapping of memory locations to banks, we can typically ignore the effect of the bank delay.
- When memory contention is high we cannot ignore the effects of bank delay. In particular modeling the delay is quite important in analyzing the performance of algorithms with high contention.
- An expansion factor beyond $\mathbf{x} = d/g$ can be used to better balance the load when using random mapping of memory locations to banks. This is particularly impor-

tant in conjunction with high memory contention. The expansion on the CRAY is such that the load is not a serious problem.

- The high-level EREW or QRQW PRAM models can be emulated on the $(d, \mathbf{x})$-BSP in a work-preserving manner as long as $\mathbf{x} \geq d/g$ and $g$ is a small constant.
- The $(d, \mathbf{x})$-BSP can serve as a bridging model between the QRQW PRAM and high-bandwidth multiprocessors. Our results show that the QRQW model is adequate for designing algorithms and generating rough predictions of running time. When necessary, predictions and implementations can be refined using the $(d, \mathbf{x})$-BSP.

There are several issues that the $(d, \mathbf{x})$-BSP does not capture, which would be needed for a more refined model of the memory system. These include the effects of the network, the effects of caching at the memory banks (available on the TERA and discussed by Hsu and Smith [30]), the effects of the order of injecting messages into the network, and any differences between the cost of reads and writes. Although the $(d, \mathbf{x})$-BSP can model local caches on processors, analyzing algorithms in this case would require that the user know which memory references are cache misses and which are cache hits, and possibly understanding traffic due to the particular cache coherence protocol.

Another area for future work is to perform similar case studies on other machines, comparing the actual performance to that predicted by the $(d, \mathbf{x})$-BSP. Finally, one could study the performance of complete applications.

# References

[1] Anant Agarwal, John Kubiatowicz, David Kranz, Beng-Hong Lim, Donald Yeung, Godfrey D'Souza, and Mike Parkin. Sparcle: An evolutionary processor design for large-scale multiprocessors. *IEEE Micro*, pages 48–61, June 1993.

[2] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith. The Tera computer system. In *Proceedings 1990 International Conference on Supercomputing*, pages 1–6, June 1990.

[3] D. A. Bader and J. JàJà. Practical parallel algorithms for dynamic data redistribution, median finding, and selection. In *Proc. 10th International Parallel Processing Symposium*, pages 292–301, April 1996.

[4] D. H. Bailey. Vector computer memory bank contention. *IEEE Transactions on Computers*, C-36:293–298, March 1987.

[5] Forest Baskett and Alan J. Smith. Interference in multiprocessor computer systems with interleaved memory. *Communications of the ACM*, 19(6):327–334, June 1976.

[6] R.H. Bisseling and W.F. McColl. Scientific computing on bulk synchronous parallel architectures. In *Proc. 133th IFIP World Computer Congress*, pages 509–514, 1994.

[7] G. E. Blelloch, M. A. Heroux, and M. Zagha. Segmented operations for sparse matrix computation on vector multiprocessors. Technical Report CMU-CS-93-173, School of Computer Science, Carnegie Mellon University, August 1993.

[8] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. A comparison of sorting algorithms for the Connection Machine CM-2. In *Proc. 3rd ACM Symp. on Parallel Algorithms and Architectures*, pages 3–16, July 1991.

[9] F. A. Briggs and E. S. Davidson. Organization of semiconductor memories for parallel pipelined processors. *IEEE Transactions on Computers*, C-26:162–169, February 1977.

[10] I. Y. Bucher and M. L. Simmons. Measurement of memory access contentions in multiple vector processors systems. In *Proceedings Supercomputing'91*, pages 806–817, November 1991.

[11] D. A. Calahan. Some results in memory conflict analysis. In *Proceedings Supercomputing '89*, pages 775–778, November 1989.

[12] L.J. Carter and M.N. Wegman. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18:143–154, 1979.

[13] Donald Y. Chang, David J. Kuck, and Duncan H. Lawrie. On the effective bandwidth of parallel memories. *IEEE Transactions on Computers*, C-26:480–489, May 1977.

[14] T. Cheatham, A. Fahmy, D.C. Stefanescu, and L.G. Valiant. Bulk synchronous parallel computing – a paradigm for transportable software. In *Proc. IEEE 28th Hawaii Int. Conf. on System Science*, January 1995.

[15] T. Cheung and J. E. Smith. A simulation study of the CRAY X-MP memory system. *IEEE Transactions on Computers*, C-35(7):613–622, July 1986.

[16] D. Culler, R. Karp, D. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *Proc. 4th ACM SIGPLAN Symp. on Principles and Practices of Parallel Programming*, pages 1–12, May 1993.

[17] D. E. Culler, A. Dusseau, R. Martin, and K. E. Schauser. Fast parallel sorting under LogP: from theory to practice. In *Proc. Workshop on Portability and Performance for Parallel Processing*, Southhampton, England, July 1993.

[18] U. Detert and G. Hofemann. CRAY X-MP and Y-MP memory performance. *Parallel Computing*, 17:579–590, 1991.

[19] M. Dietzfelbinger, J. Gil, Y. Matias, and N. Pippenger. Polynomial hash functions are reliable. In *Proc. 19th Int. Colloquium on Automata Languages and Programming, Springer LNCS 623*, pages 235–246, July 1992.

[20] M. Dietzfelbinger, T. Hagerup, J. Katajainen, and M. Penttonen. A reliable randomized algorithm for the closest-pair problem. Technical Report Research Report 513, Universitat Dortmund, December 1993.

[21] C. Engelmann and J. Keller. Simulation-based comparison of hash functions for emulated shared memory. In *Proc. Parallel architectures and languages Europe, Springer LNCS 694*, pages 1–11, June 1993.

[22] A. V. Gerbessiotis and C. J. Siniolakis. Deterministic sorting and randomized median finding on the BSP model. In *Proc. Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 223–232, June 1996.

[23] P. B. Gibbons, Y. Matias, and V. Ramachandran. Efficient low-contention parallel algorithms. In *Proc. 6th ACM Symp. on Parallel Algorithms and Architectures*, pages 236–247, June 1994. To appear in *Journal of Computer and System Sciences*.

[24] P. B. Gibbons, Y. Matias, and V. Ramachandran. The QRQW PRAM: Accounting for contention in parallel algorithms. In *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms*, pages 638–648, January 1994. Full version available as AT&T Bell Laboratories technical report, September 1994.

[25] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas. Towards efficiency and portability: Programming with the BSP model. In *Proc. Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 1–12, June 1996.

[26] J. Greiner. A comparison of data-parallel algorithms for connected components. In *Proceedings Symposium on Parallel Algorithms and Architectures*, pages 16–25, Cape May, NJ, June 1994.

[27] D. T. Harper III. Block, multistride vector, and FFT accesses in parallel memory systems. *IEEE Transactions on Parallel and Distributed Systems*, 2:43–51, January 1991.

[28] D. T. Harper III and Y. Costa. Analytical estimation of vector access performance in parallel memory architectures. *IEEE Transactions on Computers*, 42(5):616–624, May 1993.

[29] D. R. Helman, D. A. Bader, and J. JàJà. Parallel algorithms for personalized communication and sorting with an experimental study. In *Proc. Eighth Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 211–222, June 1996.

[30] W.-C. Hsu and J. E. Smith. Performance of cached DRAM organizations in vector supercomputers. In *Proc. 20th International Symp. on Computer Architecture*, pages 327–336, San Diego, CA, May 1993.

[31] J. JáJá. *An Introduction to Parallel Algorithms.* Addison-Wesley, Reading, MA, 1992.

[32] A.R. Karlin and E. Upfal. Parallel hashing—an efficient implementation of shared memory. In *Proc. 18th ACM Symp. on Theory of Computing*, pages 160–168, May 1986.

[33] R. M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A*, pages 869–941. Elsevier Science Publishers B.V., Amsterdam, The Netherlands, 1990.

[34] D.E. Knuth. *Sorting and Searching*, volume 3 of *The Art of Computer Programming.* Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1973.

[35] F. T. Leighton. Methods for message routing in parallel machines. In *Proc. 24th ACM Symp. on Theory of Computing*, pages 77–96, May 1992. Invited paper.

[36] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories. *Acta Informatica*, 21:339–374, 1984.

[37] R. Miller. A library for bulk-synchronous parallel programming. In *Proc. of the British Computer Society Parallel Processsing, Specialist Group Workshop on General Purpose Parallel Computing*, December 1993.

[38] W. Oed and O. Lange. On the effective bandwidth of interleaved memories in vector processor systems. *IEEE Transactions on Computers*, pages 949–957, October 1985.

[39] P. Raghavan. Probabilistic construction of deterministic algorithms: approximating packing integer programs. *Journal of Computer and System Sciences*, 37:130–143, 1988.

[40] R. Raghavan and J. P. Hayes. On randomly interleaved memories. In *Proceedings Supercomputing '90*, pages 49–58, November 1990.

[41] A.G. Ranade. How to emulate shared memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[42] B. Rau. Pseudo-randomly interleaved memory. In *Proceedings Int. Symp. Computer Architecture*, pages 74–83, 1991.

[43] J. H. Reif and L. G. Valiant. A logarithmic time sort for linear size networks. *Journal of the ACM*, 34(1):60–76, 1987.

[44] Subhash Saini and David H. Bailey. NAS parallel benchmark results 10-95. Technical Report NAS-95-019, NASA Ames Research Center, October 1995.

[45] B. J. Smith. A pipelined, shared resource MIMD computer. In *Proceedings International Conference on Parallel Processing*, 1978.

[46] J. E. Smith and W. R. Taylor. Accurate modeling of interconnection networks in vector supercomputers. In *Proc. International Conference on Supercomputing*, pages 264–273, Cologne, Germany, June 1991.

[47] J. E. Smith and W. R. Taylor. Characterizing memory performance in vector multiprocessors. In *Proceedings International Conference on Supercomputing*, pages 35–44, July 1992.

[48] G. S. Sohi. High-bandwidth interleaved memories for vector processors — a simulation study. *IEEE Transactions on Computers*, 42(1):34–44, January 1993.

[49] Robert H. Swendsen and Jian-Sheng Wang. Nonuniversal critical dynamics in Monte Carlo simulations. *Physical Review Letters*, 58(2):86–88, January 1987.

[50] K. Thearling and S. Smith. An improved supercomputer sorting benchmark. In *Proceedings Supercomputing '92*, pages 14–19, November 1992.

[51] Tetsutaro Uehara and Takao Tsuda. Benchmarking vector indirect load/store instructions. *Supercomputer*, VIII-6:57–74, November 1991.

[52] L.G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, 1990.

[53] M. Zagha. Effiicient irregular computation on pipelined-memory multiprocessors. Ph.D. Thesis (In Preparation), 1996.

[54] M. Zagha and G. E. Blelloch. Radix sort for vector multiprocessors. In *Proceedings Supercomputing '91*, pages 712–721, November 1991.