Work Efficient Hashing on Parallel and Vector Computers

Thomas J. Sheffler Randal E. Bryant

August 18, 1992
CMU-CS-92-172

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This research was sponsored in part by the Avionics Laboratory, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U.S. Air Force, Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Connection Machine time was provided by the Northeast Parallel Architectures Center, and by the Pittsburgh Supercomputing Center (Grant ESC910004P). Cray Y-MP time was provided by the Pittsburgh Supercomputing Center (Grant ASC890018P).

Keywords: Data-parallel, parallel algorithms, vectorized algorithms, parallel hashing

Abstract

Hashing techniques have long been used to efficiently store and locate data indexed by key. Recently, parallel hashing algorithms have been developed that allow table insertion of many keys in a few parallel steps. The analyses of these algorithms have focused on the expected number of steps required, ignoring the issue of work complexity. As a result, these algorithms have not been work efficient. In this paper, we present a parallel hashing algorithm that is shown to be work efficient because it performs no more work than its serial counterpart. An analysis of its behavior shows that it performs $S = O(\log n)$ expected parallel steps and $W = O(n)$ expected work.

Many parallel algorithms can make use of hashing as a core step. Procedures such as histogramming, set difference, keyed reduction and dictionary lookup can be formulated using a general hash routine. Thus, parallel hashing is an important fundamental parallel operation. The above applications may also be implemented using sorting as the core step. While the use of parallel hashing has been generally accepted in the folklore of parallel computing, a dearth of literature about the expected performance of such algorithms has led many to favor the use of sorting. This paper sheds light on the performance that may be achieved using parallel hashing algorithms and should lend credibility to their use.

Performance data collected from an implementation of our algorithms on the Connection Machine CM-2 and CRAY Y-MP show that it is faster than sorting by up to a factor of four on both machines. These data show that the total time required by the algorithm is more closely related to the amount of work performed, rather than the parallel step count. Our results indicate that work efficiency is an important consideration when trying to achieve high performance on parallel machines.

1 Introduction

Hashing techniques have long been used to perform table lookup on serial machines. While some hashing techniques have also been extended to parallel and vector computers [Kan90], it is only recently that the algorithmic complexity of parallel hashing algorithms has been studied[LPP91, And88]. In [Kan90], a vectorized algorithm was described that is similar to the one presented here, however no bounds were placed on the expected performance of the algorithm. While [LPP91] provided the framework for understanding the total number of parallel steps required, and [And88] provided an empirical study of parallel step counts, neither recognized the importance of work complexity. In this paper we place expected case bounds on the parallel step count of our algorithm as well as its work complexity. We show that an important indicator of actual performance is the total amount of *work* performed, and that to achieve high performance a parallel algorithm must be carefully designed to perform no more total work than necessary.

We also extend parallel hashing to the solution of a more general problem that we call the *naming* problem. Algorithms that solve the naming problem may be based on either sorting or hashing. When a total order of the elements to be named is not required, the sorting algorithm performs more work than necessary. This is borne out by the algorithmic complexities of the two methods in which general parallel sorting performs $\Omega(n \log n)$ work for arbitrary sized keys but our hashing algorithm performs only $O(n)$ expected work.

The remainder of this paper is organized as follows. First, the naming problem is defined as a generalization of hash table insertion and a number of fundamental parallel algorithms are outlined that use the naming algorithm as a core step. Next, the parallel vector model of computing is described and we present our work efficient parallel hashing algorithm. Finally, we present performance data collected from an implementation of our algorithm on the Connection Machine CM-2 and CRAY Y-MP and discuss the implications of work efficiency with regard to parallel step count. The expected case behavior of the parallel algorithm is analyzed in an appendix where we show that our parallel hashing algorithm performs no more work than its serial counterpart.

2 The Naming Problem

When building a table using hashing, each insertion of a new data item involves finding either a previous instance of the same key, or an empty site in which to put the key. A parallel algorithm for key insertion may be constructed in which a vector of keys are inserted together. The value returned by such a parallel algorithm is a vector of table indices identifying the final hash sites of each key.

This problem attempts to find a mapping of keys from a potentially large universe to a small hash table. This is an instance of the *naming* problem which is as follows: a multiset X of n elements selected from some universe U are to be assigned names in the range $\{1..m\}$, $m \geq n$, such that two elements are given the same name if and only if they are the same element from U.

The naming problem is one that is fundamental to many parallel algorithms. Often the names represent a limited resource and the keys label data that are to share those resources. In the most common instance of the naming operation, the names to be assigned are processor IDs of the available processing sites and keys identify data that are to be combined at a common site. The processing site assigned to common keys is called an *agent*, and does work on behalf of the keys assigned to it. The naming operation serves to separate the construction of a communications mapping from the operation performed with that mapping. The following list presents some examples of algorithms easily implemented using the naming operation.

Histogramming: After the keys are named, each key sends the value 1 to its agent. The agent counts the number of incoming messages using a combining send operation, which is a primitive operation of the collision+ PRAM[Wyl79].

Removal of Duplicates: The keys identify a multiset and all duplicates are to be removed. Each key sends its index in the key set to its agent. Of the indices sent to each agent, an arbitrary one is kept. Using the kept indices, the agents mark the keys to be saved, others are swept away in a packing step.

Keyed Reduction: The $\oplus$-reduction of the values associated with each key is performed by each agent processor applying a binary associative operator, $\oplus$, to all values sent to it. This is a generalization of the **histogramming** operation.

Dictionary Lookup: In this application some of the keys are dictionary entries, and others are dictionary queries. After the naming step, all dictionary entries send their values to their agent site. The keys that represent queries may then query their agent directly to find the value.

Associative Computing: The β operation of CM-Lisp [SH86] is a complex combination of communication and computation that may be expressed as a dictionary lookup step, followed by keyed reduction, followed by duplicate removal. Clearly, each of these operations may be implemented using the naming operation.

2.1 Algorithms

In many applications, there is a total order defined for the keys and the naming operation may be implemented by using a sort procedure. In this method, after the keys are placed in sorted order, they are divided into groups such that a new group begins where a key differs from its left neighbor. The first element of each group is called its "leader." The available names are distributed to the leaders, which then distribute the names to all members of their groups.

The sorting step dominates the time spent in the procedure outlined above, and may not be applicable if there is not a total order on the keys. Even when applicable, because sorting requires $\Omega(n \log n)$ operations it is not work efficient for the naming problem.

In this paper we present a family of efficient parallel algorithms that solve the naming problem by using parallel hashing. The first step of the algorithm uses a hash function, h, that uniformly distributes the n keys across the name space. At each step of the iterative procedure, a large number of keys claim names. These keys are removed from the working set by using a "packing" step so that the working set size decreases as the algorithm proceeds. Many keys find names in just the first few parallel steps, after which, the number of keys remaining in the working set decreases quickly.

3 The Parallel Vector Model of Computing

The parallel vector model of computation builds on the standard RAM model by adding a separate vector memory and vector processing unit [Ble90]. Operations on vectors have the distinction of being performed in parallel. For example, the addition of two vectors can be performed for all elements of the result vector simultaneously. A standard RAM would iterate through each of the elements to calculate the same result.

The parallel vector model imposes no restrictions on the length of a vector. Two complexity measures are introduced for algorithms in the vector model. The first is the *step* complexity, which measures the total number of vector instructions issued. Another complexity measure, the *work* complexity, measures the total number of vector elements operated on over all instructions. The work complexity of an algorithm is the same as the time complexity of the algorithm on a single processor since it measures the total number of single element operations performed. A parallel algorithm is work efficient if it has a work complexity no greater than that of an equivalent serial algorithm that solves the same problem.

The *step* and *work* complexity measures are related to the PRAM model through the following relation. For an algorithm with step complexity S and work complexity W, the PRAM complexity of the algorithm is

$$t = O(W/p + S)$$

for a p-processor PRAM. This result is due to a simulation argument in which the vector algorithm is simulated on an EREW PRAM that provides unit-step parallel-prefix operations [Ble90]. Given an unbounded number of processors, the complexity is dominated by the step complexity. Conversely, the work complexity indicates the total number of operations performed when only one processor is available. All real parallel machines have a processor count somewhere between these two extremes; when designing algorithms for the vector model it is important to keep both measures in mind.

3.1 Analysis of Algorithms

All algorithms presented in this paper will be analyzed for their step and work complexities in the vector model. In the parallel hashing algorithms the step complexity is related to the number of iterations required to complete the insertion of all n keys, while the work complexity is related to the sum of the number of active elements over all steps.

4 Parallel Hashing

Traditional serial hashing algorithms are concerned with the efficient creation of a hash table (inserting all n elements) as well as being able to locate keys already in the table [Knu68]. The naming problem only concerns itself with the insertion phase of hash table use; its final result is the set of key destinations.

Closed table hash algorithms begin with a function h that is designed to map each of the keys uniformly to some number in the range $\{1..m\}$ for the given m. As each key is inserted, its hash location is probed to examine whether another key has already been inserted there. If so, an iterative search procedure is used to find an empty site. The sequence of hash sites examined by each key is called a "path." Ideally, each key examines a different path so as to avoid a situation where many keys collide repeatedly along the same path. This phenomenon is called "clustering."

In the parallel version of hashing, the insertion of many keys is attempted in one step. Those that fail all examine the next site in their path in parallel. The process continues until all keys find a hash site.

Whereas the analysis of the running time of the serial algorithm is concerned with the *average* time of insertion for each of the n keys, the parallel algorithm must consider two measures of performance. The first is the total iteration count, which is governed by the *maximum* number of probes required to insert any of the keys since the algorithm cannot terminate until all keys find a hash site. The second measure is the total work performed, which is the sum over all iterations of the lengths of the vectors involved in each operation.

Our algorithm accepts a vector of n *keys* and returns a vector of *names*. A hash table of length m is allocated and initialized so that all locations have the special value EMPTY. Upon completion, the name associated with *key[i]* is located at *name[i]*. On the CM-2, the name for each key is found on its own processor at the completion of the algorithm.

Pseudo-code for our algorithm is presented below. In the initialization step (Lines 1–4), the keys are copied to a vector called *active* and the initial hash sites are calculated by **hash-fn**. The *home* vector records the starting location of each key. After each iteration of the algorithm, these temporary vectors are adjusted to hold only the information relevant to those keys that have yet to find a hash site.

The main step of the algorithm is described by the parallel-do loop of Line 6. In parallel, all keys check to see if their hash site is empty. Those that find an empty site try to claim it by sending their key there. When multiple keys are sent to the same site, one overwrites all others ensuring that at least one key claims the site. With one more table reference, the hash site of each key is examined to determine which key claimed it. Keys that find their own value are marked done and their hash site is sent *home* to the *names* vector. All other keys continue with further probe steps by calculating a new hash site in Line 18.

```
     input:      keys[n], table[m];
     output:     names[n];
     temporary:  hash[n], get[n], active[n], home[n], notdone[n];
     procedure parallel-insert {
(1)      pardo (0 ≤ i < n){                          (* Initialization *)
(2)          active[i] = keys[i]
(3)          home[i] = i;
(4)          hash[i] = hash-fn(keys[i], m);
         }
(5)      while (length(active) ≠ 0) {
(6)          pardo (0 ≤ i < length(active)) {
(7)              get[i] = table[hash[i]];            (* check hash site *)
(8)              if (get[i] == EMPTY) {
(9)                  table[hash[i]] = keys[i];       (* attempt to claim it *)
(10)                 get = table[hash[i]];           (* check it again *)
                 }
(11)             notdone[i] = (get[i] ≠ keys[i]);
(12)             if (not(notdone[i]))
(13)                 names[home[i]] = hash[i];
             }
(14)         active = pack(active, notdone);         (* pack unfinished keys *)
(15)         home = pack(home, notdone);
(16)         hash = pack(hash, notdone);
(17)         pardo (0 ≤ i < length(hash)) {
(18)             hash[i] = next-fn(hash[i], active[i]);   (* probe step *)
             }
         }
```

The use of the **pack** step on Lines 14–16 ensures that this procedure is efficient by producing a new vector that holds only the elements that are flagged as *notdone*. Procedure **pack** is implemented by first enumerating the **true** sites and then sending the values at these sites to their enumeration address. In the parallel vector model it is implemented with a single "scan" operation (parallel-prefix) and a parallel vector permutation. Thus, its complexity is $S = O(1)$ parallel steps and $W = O(n)$ work for a vector of length n.

The function **next-fn** may be designed to implement a number of different probe strategies. The path that each key follows may be defined by either "Linear Probing" or "Double Hashing" probe strategies. Linear probing simply searches the hash sites in sequence. Double hashing attempts to reduce "clustering" by using a probe strategy where the path each key follows is a function of its value. To implement this algorithm, a second hash function, h_2, is employed and the ith site examined for the key is $((h(A) + ih_2(A)) \bmod m)$.

Another variation also adopted in many serial hashing algorithms is the use of an "Overflow" hash table. In this approach, two hash tables (designated T_1 and T_2) are used such that each key makes its first probe into T_1 and makes all other probes in T_2. This arrangement takes advantage of the fact that many keys require only one probe, leaving table T_2 nearly empty.

In the appendix, we show that for Linear Probing, our algorithm performs a total of $S = O(\log n)$ expected steps, and $W = O(n)$ expected work. Analyses of double hashing algorithms show that on the average, they reduce the total number of probe sites examined [Knu68]. We implemented a version of double hashing and compare our results to linear probing on the CM-2. On the CRAY we implemented both probe strategies with and without an overflow table. The performance of our implementations is discussed in the following sections.

5 Implementation on the Connection Machine CM-2

We implemented our algorithms on the Connection Machine model CM-2. Through the virtual processor (VP) mechanism of the system software we were able to support the varying sized vectors that our **pack** procedure produced. As stated earlier, this was necessary to ensure work efficiency.

Our choice of a hash function was guided by the target architecture for implementation. One particularly good hash function is based on a result from coding theory [Gal68]. It treats an n-bit key as a polynomial in $\mathrm{GF}(2^n)$ and computes an m-bit hash value as the remainder of division by a primitive irreducible polynomial of the field $\mathrm{GF}(2^m)$. These hash functions are often not used on conventional architectures because of the bit manipulations that must be performed. Because the CM-2 is a bit-serial machine, these hash functions are extremely fast. Also, because these hash functions require a table whose size is a power of two, they are perfectly suited to the size of VP sets and guarantee that keys are evenly distributed over all processors of the CM-2.

A minor modification was made to the insertion procedure. Because communications operations require as much as 1000 times as much time as elementwise operations on the CM-2, the procedure **parallel-insert** was modified to reduce the total number of communications operations performed by omitting the first **get** step. Instead, as hash sites are claimed, a flag is set. During the probe step, all keys send their values regardless of the state of their hash site. Then, non-empty hash sites overwrite the value sent there with the key that previously claimed the site. Even with large hash tables, these elementwise operations take virtually no time, and are outweighed by the savings from eliminating the extra communications operations. Theoretically, this increases the work complexity of the algorithm by $O(m \log n)$, but does not have an effect when $\log n < 1000$, which is a reasonable assumption for our problem sizes.

5.1 Results

A series of trials were taken for n ranging from 50,000 to 2,000,000 elements on a 16k processor Connection Machine. The key size in this trial was 32 bits but the universe of possible keys was restricted to be a random set of size n. In this manner, the keys represent a random mapping of a set into itself. Each trial point was repeated a total of five times and the average time reported

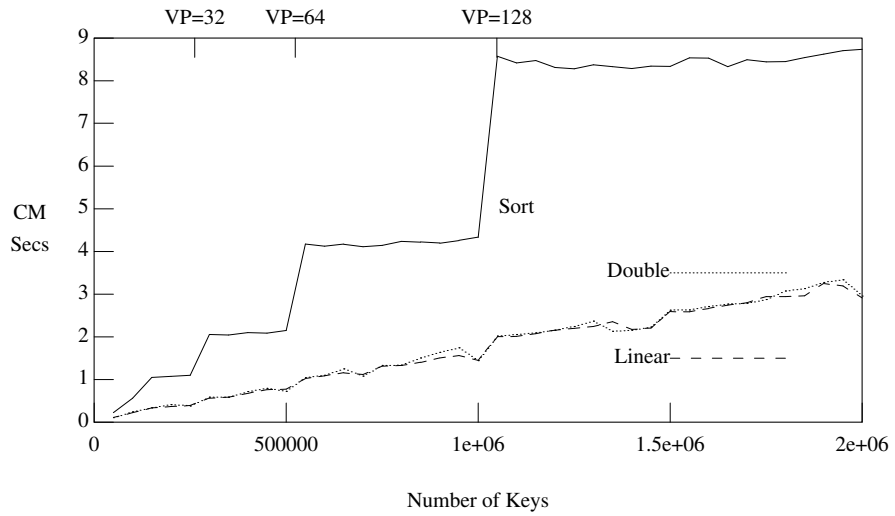

Figure 1: A comparison of the hashing algorithms to a naming algorithm that uses sorting. Through all trials the hashing algorithms performed better by a factor of 3 to 4.

was measured in CM seconds, the amount of time the Connection Machine required to complete all iterations. Both "linear probing" and "double hashing" schemes were compared to a naming algorithm that sorted the keys using the RANK function of the system software and enumerated the leaders to assign names. Figure 1 summarizes our results.

Through all trials, the hash based algorithms outperformed the sorting based algorithm by a factor of 3 to 4. The sharp jumps in times for the sort based algorithm occurred where the vector size increased past a VP ratio boundary. Our data spans VP ratios of 2 to 128 and shows that the hashing algorithms have linear characteristics even over VP ratio boundaries.

It is interesting to notice that the two variants of hashing require almost the same time for all key set sizes. Even though the total number of iterations required by the "linear probing" variant *always* exceeded those required by "double hashing", the extra iterations were performed on such small vectors that the additional time is negligible. This point deserves further examination.

Figure 2 shows the average number of iterations required for each algorithm for different input set sizes. The sharp drops in the curves are due to the fact that the hash table and overflow table sizes were rounded to the next power of two when the key set size exceeded a VP ratio boundary. In those situations when the hash table was very large with respect to the number of keys, the number of iterations became quite small. While we see that double hashing keeps the number of iterations performed smaller than by using the linear probing method, the total times spent by the algorithms are roughly the same, and are not affected very much by the occasional large jumps in the number of iterations.

Figure 3 provides further insight into the impact of the total number of iterations on the completion time. The trials for the insertion of 1,900,000 keys using double hashing required a wide range of iterations to complete. Even as the iterations increased by almost a factor of 2, the total time grew by only 7%. The time is dominated by the first few probe iterations with long vectors; all other iterations are performed with very short vectors and require a small percentage of the total time.

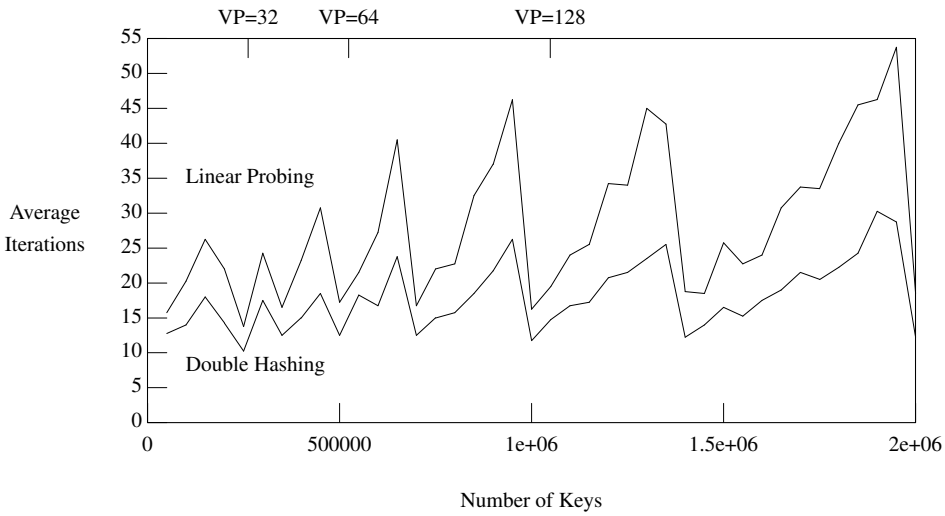

Figure 2: The average number of iterations required by the two hashing algorithms. The data show that double hashing keeps the total number of iterations lower than the number required by linear probing. The unusual drops in the curves occur when the hash table increased to the next VP set size on the CM, resulting in a hash table with a very low load factor.

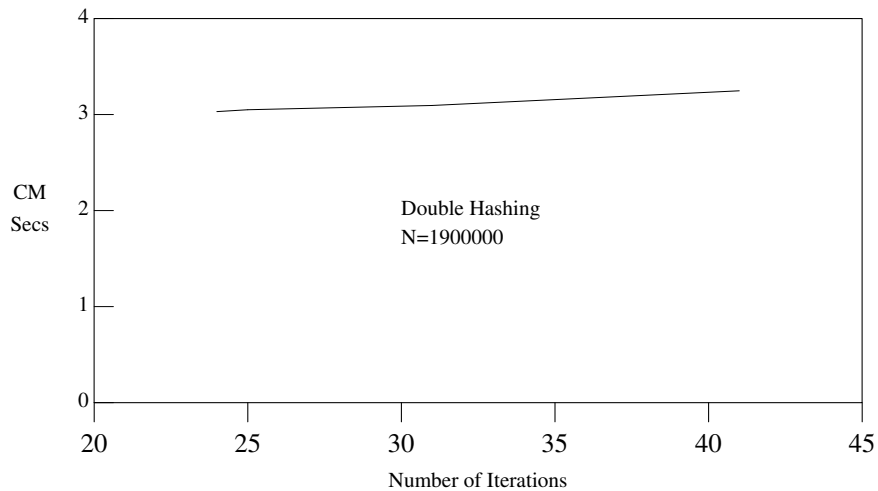

Figure 3: A comparison of the execution times when the number of iterations varied greatly. Even though the number of iterations spanned a wide range, the time increased by only a small amount when the number of iterations grew large. All of the extra iterations were performed with very small vectors and required very little time.

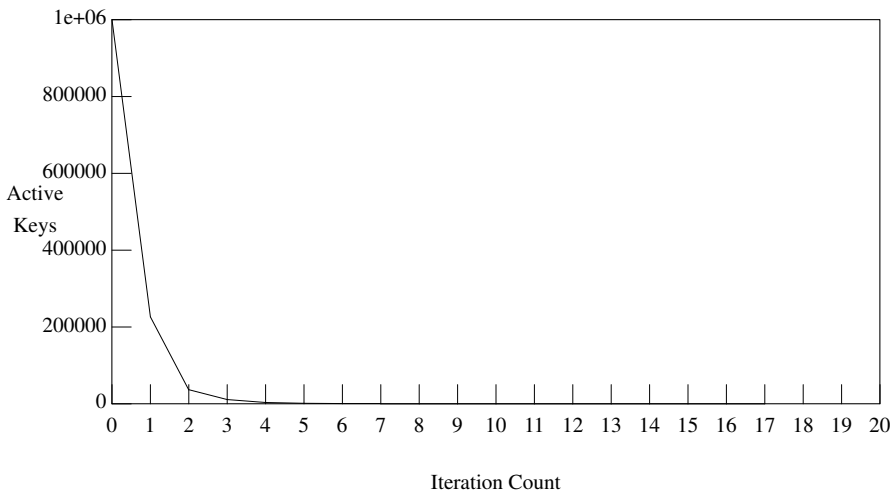

Figure 4: The number of active keys at each step in the hashing algorithm. The hashing algorithm eliminates many keys from the active set in just a few iterations. In this graph, the total number of active keys is less than 100 after eight iterations. Most of the iterations are performed with very short vectors.

The decrease in the number of active keys is shown in Figure 4. When inserting one million keys using the double hashing algorithm, by the eighth iteration the total number of active keys fell below 100. Probe iterations performed with small vector lengths are completed quickly.

6 Implementation on the CRAY Y-MP

The implementation of the hashing algorithm on the CRAY Y-MP was straightforward because all operations completely vectorize. We used the CVL (C Vector Library) developed by Guy Blelloch at Carnegie Mellon University for most of the vector operations [Ble92], but developed our own hash function in CRAY vectorized C.

As with the implementation for the CM-2, the hash function was highly optimized for the target architecture. We chose to use a standard remainder operator as our hash function and sized the hash tables to be the least prime number greater than the number of keys. However, because the standard modulo operator ('%') of CRAY C calls a subroutine that computes the remainder between two 64-bit integers, it was much too slow. We wrote a highly optimized modulo function that took advantage of the facts that the same modulus was applied to all keys, and that it had far fewer than 64-bits of significance. Using these observations, we divided the computation into two parts that each used an integer division and a subtraction operation, yielding a very fast hash function.

The next-site function, **next-fn**, was written to allow the choice of Linear or Double probing, with or without an overflow table. Double probing was described earlier. To implement an overflow table, we conceptually divided the hash table into two parts: it's lower half acting as T_1, and the upper half acting as T_2. The next-site function was written to supply a first probe site in the lower half, with all other probes indexing sites in the upper half. In this manner, a single site address

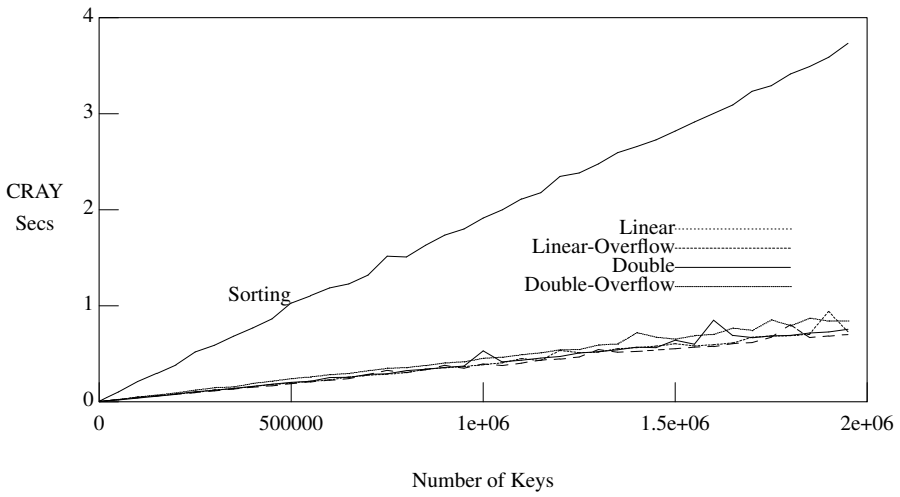

Figure 5: A comparison of the times of the hashing algorithms to the time to sort the keys on the CRAY Y-MP. Through all trials, the hashing algorithms performed better by a factor of 4 to 5. While the more complicated probe strategies required fewer total iterations, the total time required by each was nearly equal.

could specify one of the two tables and a site within that table.

6.1 Results

Once again, we measured the performance of our hashing algorithm relative to sorting. This time, the sorting time measured represents *only* the time required to sort the keys using the ORDERS subroutine of the system software. The key size in this trial was 64 bits and the keys once again represented the random mapping of a set into itself. Figure 5 summarizes the performance for each of the four combinations of probe strategies relative to sorting. Throughout all trials, the hashing algorithms outperformed the CRAY ORDERS subroutine by up to a factor of 5.

It is interesting to notice that the total times required by each of the four approaches to hashing are nearly equal. In general though, the fastest approach was the simple Linear Probing search, while the slowest was Double Hashing with an Overflow table. These results are somewhat surprising in light of the total number of iterations required by each approach.

Figure 6 displays the average number of iterations per key required by each of the variations. Clearly, the simple Linear Probing approach performs far more steps than when using Double Hashing with an Overflow Table. The effect observed here is that the extra complexity of the next-site function (**next-fn**) when using Double Hashing and an Overflow table far outweighs any benefits achieved by a reduction in the number of iterations. Thus, the simple and straightforward Linear Probing approach is the fastest even with the cost of additional iterations. While the more sophisticated probing algorithms did reduce the total number of probe steps required, their extra complexity caused a loss in actual performance.

This effect is due to the fact that computation and communication are so nearly balanced on the CRAY Y-MP. While some machines perform local arithmetic operations much faster than they can permute a vector, on the CRAY Y-MP these operations require nearly the same amount

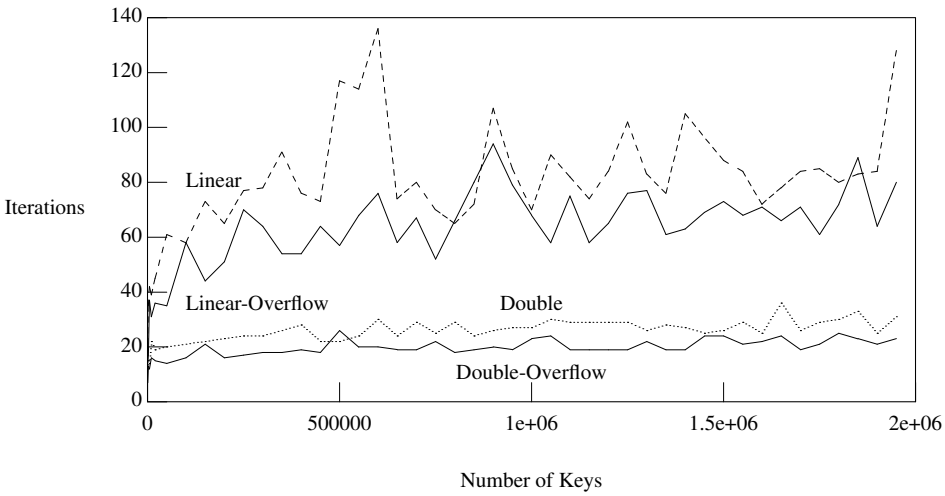

Figure 6: The average number of iterations per key on the CRAY Y-MP for each of the four approaches. In general, the more complicated probing schemes reduced the total number of probe steps, as expected. However, as the complexity of the probe function increased, so did the execution time, making the simplest probe scheme preferable.

of time per element. Because of this, it is faster to perform the simple Linear Probing hashing algorithm than to use a more clever (and complicated) probing function. On a machine with slower communication, it might be wise to use the more sophisticated probing schemes.

7 Conclusions

The naming problem is one that is fundamental to many parallel algorithms. While the most straightforward implementation of this parallel kernel uses sorting to organize the keys, we have presented an efficient randomized hashing-based algorithm that is more efficient both in theory and practice.

The main idea that should transfer from this work is the importance of reducing the problem size as the algorithm progresses. Many Connection Machine algorithms have been designed with a fixed number of processors in mind. This design influence reflects an earlier view of the CM when the number of virtual processors was fixed throughout the execution of a program. Because virtual processor sets may be allocated dynamically, the view of the CM is one of a dynamically reconfigured vector processor, whose length adjusts to fit the problem size. We have shown the importance of reducing the problem size and using smaller VP sets when possible. On vector computers such the CRAY Y-MP, the benefit of *compressing* vectors has long been recognized.

Work efficiency for parallel algorithms has often been sacrificed in favor of reducing the total number of parallel steps an algorithm requires. This paper shows that for some algorithms, variations in the total number of parallel steps can have a small impact on the actual performance and that work complexity is an important measure.

8 Acknowledgments

We would like to thank Guy Blelloch for the many helpful comments he had throughout the course of this work.

References

[And88] Paul B. Anderson. Parallel hashed key access on the connection machine. In Ronnie Mills, editor, *Second Symposium On The Frontiers of Massively Parallel Computations*, pages 643–645. IEEE, IEEE Computer Society Press, 1988.

[Ble90] Guy E. Blelloch. *Vector models for data-parallel computing.* MIT Press, 1990.

[Ble92] Guy E. Blelloch. NESL: a nested data-parallel language. Technical Report CMU-CS-92-103, School of Computer Science, Carnegie Mellon University, 1992.

[Gal68] R. G. Gallagher. *Information Theory and Reliable Communication.* John Wiley and Sons, Inc., 1968.

[Gon90] Gaston H. Gonnet. *Handbook of Algorithms and Data Structures.* Addison Wesley, 1990.

[Kan90] Yasusi Kanada. A vectorization technique of hashing and its application to several sorting algorithms. In *PARBASE-90, International Conference on Databases, Parallel Architectures, and Their Applications*, 1990.

[Knu68] Donald Knuth. *The Art of Computer Programming; Volume 3: Sorting and Searching.* Computer Science and Information Processing. Addison-Wesley, 1968.

[LPP91] Fabrizio Luccio, Andrea Pietracaprina, and Geppino Pucci. Analysis of parallel uniform hashing. *Information Processing Letters*, 37:67–69, January 1991.

[Pet57] W. W. Peterson. *IBM J. Research and Development*, 1:130–146, 1957.

[SH86] Guy L. Steele Jr. and W. Daniel Hillis. Connection Machine Lisp: Fine-grained parallel symbolic processing. Technical Report 86.16, Thinking Machines Corporation, May 1986.

[Wyl79] James Christopher Wyllie. *The complexity of Parallel Computation.* PhD thesis, Cornell University, 1979.

A Analysis of the Algorithm

The behavior of the parallel hashing algorithm for linear probing will be analyzed by comparing it to its serial counterpart. The total work, W, of the parallel hashing algorithm is the sum of the lengths of the work vector over all iterations. Since each key remains in the active vector for only as many iterations as it requires probe steps, the total work is the sum of the number of probes required to insert all n keys. When the hash function and hash table size are the same for the serial and parallel algorithms, the total number of probes required by each of the algorithms is shown to be the same.

The number of iterations required to insert all of the keys could conceivably be n for a particularly bad arrangement of keys, however this is almost never the case. An expected bound on the number of iterations required will be made by examining the expected number of probes required to insert the worst-case key.

A.1 The total work required to insert N keys

A surprising property of the serial algorithm for Linear Probing was first pointed out by W. W. Peterson [Pet57]:

THEOREM 1: The total number of probes required to insert N keys remains unchanged regardless of the order in which the keys are inserted in the table.

PROOF: (due to Knuth, [Knu68]) The keys are presented in some order $A_1, A_2, ...A_N$. It suffices to show that the total number of probes needed to insert the keys is the same as the total number needed for $A_1...A_{i-1}A_{i+1}A_iA_{i+2}...A_N, 1 \leq i < N$. There is clearly no difference unless the $(i+1)$st key in the second ordering falls into the position occupied by the ith in the first ordering. But then the ith and the $(i+1)$st merely exchange places, so the total number of probes for the $(i+1)$st is decreased by the same amount the number for the ith is increased. ∎

A similar theorem holds for inserting keys in parallel.

THEOREM 2: The total number of hash sites examined remains unchanged if two or more keys are inserted in parallel.

PROOF: Consider two keys presented in order A_1, A_2. If, in parallel, the two keys begin their probes at two different locations, then because they each move only one site forward each iteration, the two keys will find the same empty hash sites as they would had they been inserted in sequence. If the keys happen to begin at the same site, then the first two empty sites past their starting point are the two sites that the keys will find. The two possible arrangements of these two keys into the two sites are exactly the two arrangements dictated by the serial presentation order A_1, A_2 and A_2, A_1. Thus, parallel insertion has the same effect as a random exchange of the order in which keys are inserted by a serial algorithm, which by the previous theorem does not affect the total number of probe steps made. ∎

These two theorems state that the parallel linear probing algorithm performs exactly the same number of probes as its serial counterpart. The analysis of the numbers of probes required to insert n elements into a hash table using linear probing is well understood. From [Knu68] we have the following: The average number of probes required to insert the ith key is $C_i' = \sum_{1 \leq r \leq m} rP_r(i)$ where $P_r(i)$ is the probability that the ith key requires r probes to find an unoccupied site. The average number of probes to insert all n keys is $C_n = \frac{1}{n}\sum_{0 \leq i \leq n} C_i'$. The total work is simply $W = nC_n$ for n keys. Knuth gives the value of C_n for linear probing as $C_n \approx \frac{1}{\alpha}\log\frac{1}{1-\alpha}$ where $\alpha = \frac{n}{m}$. Thus, when the hash table is sized so that $m = \Theta(n)$, the expected work performed by the parallel algorithm for linear probing is

$$W = nC_n \approx n\frac{1}{\alpha}\log\frac{1}{1-\alpha} = O(n).$$

A.2 The total number of iterations required to insert N keys

The Longest Length Probe Sequence (LLPS) quantity associated with a hashing algorithm measures the longest sequence of probes needed to locate any of the n keys inserted in the table. The LLPS measure is a random variable whose maximum value is obviously n for linear probing. However its average value over all possible arrangements of keys in the hash table, the average LLPS, is also of interest.

Since the parallel linear probing algorithm requires as many iterations as the longest length probe sequence for any key, the average LLPS figure of the serial algorithm measures the average number of iterations that our parallel hashing algorithm requires. Gaston Gonnet [Gon90] has derived bounds on the average LLPS for linear probing and shows that it is $O(\log n)$, when $m > n$.

Thus, the parallel step complexity for n keys is

$$S = O(\log n).$$

Our experience shows that the average number of iterations required for the hashing algorithm to complete grows slowly with the number of keys. Data presented later shows that the insertion of one million keys is achieved in less than 20 iterations, on average. However, the work figure is a more accurate measure of the total amount of time required by the algorithm.