PARALLEL SEARCHING FOR A FIRST SOLUTION[1]

Bożena BARTOSZEK[a], Zbigniew J. CZECH[b] and Marek KONOPKA[c]

[a]Computing Laboratory, University of Kent, Canterbury, CT2 7NF, England[2]
[b]Institutes of Computer Science, Silesian Technical University and
Polish Academy of Sciences, ul. Bałtycka 8, 44-100 Gliwice, Poland
[c]Institute of Computer Science, Silesian Technical University, ul. Pstrowskiego 16,
44-100 Gliwice, Poland

Abstract. A parallel algorithm for conducting a search for a first solution to the problem of generating minimal perfect hash functions is presented. A message-based distributed memory computer is assumed as a model of parallel computations. A data structure, called reversed trie (r-trie), was devised to carry out the search. The algorithm was implemented on a transputer network. The experiments showed that the algorithm exhibits a consistent and almost linear speedup. The r-trie structure proved to be highly memory efficient.

1. INTRODUCTION

Consider a static[3] set W of m finite length words over an ordered alphabet. A *hash function* is a function $h : W \to I$ that maps the set W into some given interval of integers I, say $[0, k-1]$, where $k \geq m$. The hash function computes for each word from W an address (an integer from I) for the storage and retrieval of that word. If $k = m$ and h is an injection, then we say that h is a *minimal perfect hash function* (MPHF). Usually for a given set W many MPHFs exist. We are interested in finding only one of these functions.

MPHFs are used for memory efficient storage and fast retrieval of items from a static set, such as reserved words in programming languages, command names in operating systems, commonly used words in natural languages etc.

Various algorithms for constructing MPHFs have been proposed. Many of them involve an exhaustive search which terminates when a first solution is found. In this paper we present a parallel algorithm for conducting such a search. We assume a message-based distributed memory computer as a model of parallel computations.

The paper is organized as follows. Section 2 contains an outline of the sequential algorithm which was a basis for our work. In section 3 we give an overview of some parallel search

[1]M. Konopka and Z. J. Czech were supported by the Polish Committee for Scientific Research under the grant BK-608/RAu2/93. B. Bartoszek was supported by the European Committee under the Tempus grant.

[2]On leave from the Silesian Technical University.

[3]By static we mean a set that is essentially unchanging, i.e. it is not subject to insertions or deletions of elements.

algorithms. In section 4 we present our parallel algorithm and two implementations. Section 5 describes the experimental results, and section 6 contains conclusions.

2. A SEQUENTIAL ALGORITHM FOR FINDING MPHF

Czech and Majewski [1] proposed a linear time algorithm for finding MPHFs. It searches for the MPHF of the form:

$$h(w) = (h_0(w) + g(h_1(w)) + g(h_2(w))) \text{ mod } m$$

where h_0, h_1 and h_2 are auxiliary pseudorandom functions, and g is a function implemented as a lookup table, whose values are established during the exhaustive search.

The MPHF is constructed in three steps. First, the *mapping* step transforms a set of words into a set of triples of integers h_0, h_1 and h_2. The second step, *ordering*, divides set W into subsets W_0, W_1, …, W_k, such that $W_0 = \emptyset$, $W_i \subset W_{i+1}$, and $W_k = W$, for some k. The sequence of these subsets is called a *tower*, each subset $X_i = W_i - W_{i-1}$ is called a *level* of the tower, and k is called a *height* of the tower. The third step, *searching*, tries to extend the function h from the domain W_{i-1} to W_i for $i = 1, 2, \ldots, k$.

Observe that allocating a place in the hash table for a word w requires selecting the value $U(w) = g(h_1(w)) + g(h_2(w))$. There may exist a sequence of words $\{w_0, w_1, \ldots, w_{j-1}\}$, such that $h_1(w_i) = h_1(w_{i+1})$ and $h_2(w_{i+1}) = h_2(w_{(i+2) \text{mod}\, j})$, for $i = \{0, 2, 4, \ldots, j-2\}$. Once words w_0, w_1, …, w_{j-2} are allocated some places in the hash table, both $g(h_1(w_{j-1}))$ and $g(h_2(w_{j-1}))$ are set. Hence, word w_{j-1} cannot be allocated an arbitrary place, but it must be placed in the hash table at location

$$h(w_{j-1}) = (h_0(w_{j-1}) + U(w_{j-1})) \text{ mod } m.$$

In the sequence the words w_0, w_1, …, w_{j-2} are independent (i.e. they have a choice of a place in the hash table), whereas the word w_{j-1} is dependent (i.e. it has not such a choice). These words are called in [1] *canonical* and *noncanonical*, respectively. It is easy to see that

$$U(w_{j-1}) = g(h_1(w_{j-1})) + g(h_2(w_{j-1})) = \sum_{p \in path(w_{j-1})} (-1)^p U(w_p)$$

where $path(w_{j-1})$ is a sequence of words $\{w_0, w_1, \ldots, w_{j-2}\}$, and thus

$$h(w_{j-1}) = (h_0(w_{j-1}) + \sum_{p \in path(w_{j-1})} (-1)^p U(w_p)) \text{ mod } m.$$

If the place $h(w_{j-1})$ is occupied, a collision arises and no MPHF for selected values of g can be found.

In the searching step the following combinatorial problem is solved:
find $U(w_i) \in [0, m-1]$, $i = 1, 2, \ldots, k$, where k is the height of the tower, such that values $h(w_i) = (h_0(w_i) + U(w_i)) \text{ mod } m$ for canonical words $w_i \in W$, and $h(w_j) = (h_0(w_j) + \sum_{p \in path(w_j)} (-1)^p U(w_p)) \text{ mod } m$ for noncanonical words $w_j \in W$ are all distinct, i.e. for any w_1 and $w_2 \in W$, $h(w_1) \neq h(w_2)$. The $U(w_i)$s (or U-values) are found during the exhaustive search at every level X_i of the tower. The search starts with $U(w_i) = 0$ for each canonical word

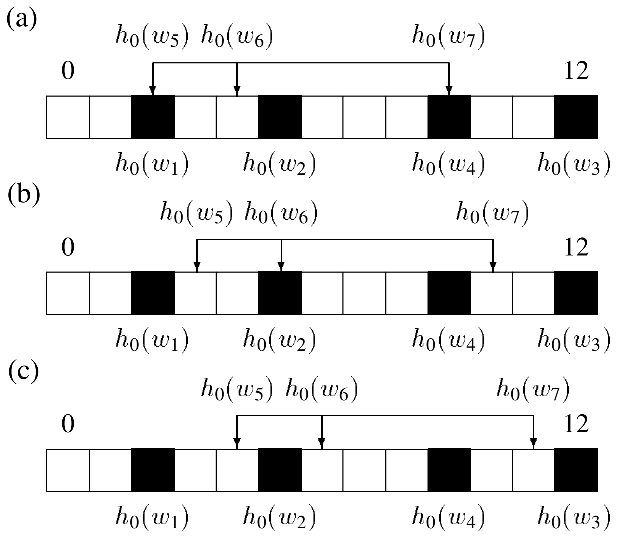

Figure 1: Fitting the pattern in the hash table

w_i, i.e. an attempt is made to locate it at the position $h_0(w_i)$ in the hash table. Once the hash value for the canonical word w_i on a given level of the tower is found, the value of $U(w_i)$ is known. It enables to compute the hash values for the noncanonical words on that level. The set of the hash values of words on a given level X_i of the tower is called a *pattern*. If all places defined by the pattern are not occupied, the task on a given level is done and the next level is processed. Otherwise, the pattern is moved up the table modulo m until the place where it fits is found. Except for the first level of the tower, this search is conducted when the table is partially filled. Thus, it may happen that no place for the pattern is found. In such a case the searching step backtracks to earlier levels, assigns different hash values for words on these levels, and then again recomputes the hash values for successive levels.

Example 1. Let tower T consists of the following sets (levels): $X_1 = \{w_1\}$, $X_2 = \{w_2, w_3\}$, $X_3 = \{w_4\}$, $X_4 = \{w_5, w_6, w_7\}$. Assume that the first words specified in each set are canonical. Let h_0 values be:

w_i	w_1	w_2	w_3	w_4	w_5	w_6	w_7
$h_0(w_i)$	2	5	12	9	2	4	9

There is no problem with placing the first three levels of T in the hash table by setting $U(w_1) = U(w_2) = U(w_3) = 0$. The black boxes in Fig. 1a show the assigment of addresses to words w_1 to w_4. The words on level 4 form the pattern $(h_0(w_5), h_0(w_6), h_0(w_7)) = (2, 4, 9)$. This pattern cannot be placed in the table with $U(w_5) = 0$ so we move it up the table, first by one (Fig. 1b) and then by two positions (Fig. 1c) where it finally fits. Thus, $U(w_5) = 2$. □

In [2] Czech and Majewski proposed *backtrack pruning* to speed up the search. Consider two words w_a and w_b on level i of the tower for which $h(w_a) = (h_0(w_a) + \sum_{p \in path(w_a)} (-1)^p U(w_p)) \bmod m = h(w_b) = (h_0(w_b) + \sum_{q \in path(w_b)} (-1)^q U(w_q)) \bmod m$. Let

$P = \{w_p \mid p \in path(w_a)\}$ and $Q = \{w_q \mid q \in path(w_b)\}$, and let *level*($w$), $w \in P \cup Q$, be a function that returns a tower level of w. After having discovered a conflict for words w_a and w_b instead of decreasing i by 1, the search is continued on level $d = \max\{\ \textit{level}(w) | w \in P \cup Q\}$, such that if $w \in P \cap Q$, *level*$(w) = d$, then the $U(w)$ values have different signs in $h(w_a)$ and $h(w_b)$.

As we have already mentioned, although there can be many solutions to the problem of generating MPHFs, we are interested in finding any one of these solutions.

The exhaustive search applied in the third step of the algorithm has a potentially worst-case time complexity exponential in the number of words to be placed in the hash table. The time to perform the search depends on the size of table g. It has been proved that if $|g| = 2m$, then the time to perform the search is negligible and is dominated by the time to perform the mapping and ordering steps [1]. However, when the table size is decreased, the time taken by the search grows exponentially (see Table 1). Since the g table contains the description of the MPHF, it is desirable for g to be as small as possible.

3. PARALLEL SEARCH ALGORITHMS

The exhaustive search can be formulated as a search of a state-space usually structured as a tree. A single state, or node, of the tree is transformed to successor states by using operators. These transformations correspond to arcs between nodes and their successors in the tree. The simplest approach to parallelization of the search is to apply at any node the operators to obtain successor nodes, and then to search the subtrees beneath each node in parallel by different processors. However, if we search only for a single solution and it is found in the first subtree, the work done in the other subtrees is wasted. As a consequence, such a parallel search can give a sublinear speedup or even a decrease in speedup with an increase in the number of processors. It is also possible to obtain a superlinear speedup, when for example a solution is found relatively quickly in the *p*-th subtree, where *p* is the number of processors. These anomalous results were first observed and discussed by Lai and Sahni [3].

Kalé and Saletore [4] suggested two criteria to evaluate parallel search schemes:

(1) The time required to find a solution. A scheme must be able to *consistently* generate a solution faster than the best sequential scheme. The speedup should be possibly close to p, and must increase monotonically with the addition of processors.

(2) The amount of memory required to conduct the search. This amount depends on the search scheme and may vary from linear to exponential function of the depth of the tree.

In [4] Kalé and Saletore proposed a priority based parallel depth-first search. The idea is to mimic the behavior of the sequential algorithm and to search the tree left-to-right. As a result, the wasted work done to the right of the first solution is minimized, which is desirable if a consistent linear speedup is to be achieved. The *priority bit-vectors*, defined as the sequences of bits of arbitrary length, are used. Priorities which are compared lexicographically are dynamically associated with nodes when they are created. A zero-length priority vector is associated with the root of the search tree. The priority of a descendant node is obtained by appending its number — given by the ranking of descendant nodes from left-to-right — to the priority of its parent. The active nodes are kept in a shared priority queue defining the order in which the nodes are searched.

The worst-case queue length for the prioritized search is $O(pdb)$. This length can be reduced by making use of two techniques: *binary decomposition* and *delayed-release* [4]. We describe shortly the latter one, as it is adopted in our algorithm. The idea behind the delayed-release technique is that when a node is expanded, the descendants are not immediately available to other processors. Only the first descendant is inserted into the queue. This is applied to every node until a leaf node is reached. Then, all the nodes are released as active nodes and may be picked up by other processors. As a result, the intermediate levels of the tree are skipped and the tree is explored from its bottom.

The authors claim [4] that those techniques virtually eliminate the wasted work and decrease the demanded memory to the amount of $O(p + d)$.

4. AN R-TRIE BASED PARALLEL SEARCH

In our parallel search algorithm we use the technique of the processor farm. The basic concept of farming consists of having a central controller — the *Master* process — that hands out pieces of work to be processed by the members of a pool of *Worker* processes. The *Master* stores the active nodes of the search tree that represent the partial solutions to the MPHF problem. A solution is defined by a sequence of *U*-values, one value for each level of the tower. The nodes are sent to the *Workers* which do the search by placing successive levels of the tower in the hash table and reporting the results to the *Master*.

The partial solutions are kept in a specially devised data structure that we called a *reversed trie* (*r-trie*). The name comes from its resemblance to the *trie* structure discussed by Knuth in [5]. An r-trie is essentially a *b*-ary tree, whose nodes correspond to the *U*-values. The nodes on level l represent the set of partial solutions that begin with a certain sequence of *U*-values. In a single node of an r-trie we hold: a level number, a *U*-value for that level, a flag indicating whether the node is assigned to a *Worker* or not, and a pointer to the parent node. We denote a node by a pair (level number, *U*-value), and (1, 0) is the root of an r-trie. A partial (or complete) solution is restored by traversing the nodes of an r-trie from a leaf up to its root. The active nodes in an r-trie are arranged into a doubly-linked list named *ActiveNodes*. This list is used for selecting nodes and handing them out to *Workers*.

Example 2. Fig. 2b shows the r-trie. The nodes on level 3, e.g., represent partial solutions given by the following sequences of *U*-values: (0, 5, 2) and (0, 5, 3). □

The advantage of the r-trie structure over the approach by Kalé and Saletore is that we do not need to compute priority vectors. Also, no priority queue has to be maintained. To find a node to assign for a *Worker* we scan the *ActiveNodes* list inspecting at most $p - 1$ of its elements.

4.1. Parallel-C implementation

4.1.1. *Worker* processes

A *Worker* process receives two types of messages from the *Master*:

New. This message contains a new partial solution for carrying on the search. It consists of a level number where the new partial solution begins, a number of levels in the solution, and the *U*-values, one value for each level.

Continue. This message has no components. Upon receiving the *Continue* message a *Worker* continues its search with the currently available partial solution.

Let *i* be the number of the last level placed by a *Worker*, and let the *New* message it received from the *Master* contain the *n*-level partial solution beginning from level *j*. (Since the *Master* keeps track of the progress of each *Worker*, $j \leq i$ always holds.) In response, the *Worker* frees all the places in the hash table occupied by the words on levels $j,\ j+1,\ \ldots,\ i$. Then it places the words from levels $j,\ j+1,\ \ldots,\ j+n-2$ by making use of the *U*-values received, and continues the search from level $j+n-1$. Denote this level as k_0. The idea of delayed-release is adopted in our algorithm. Thus, placing of successive levels of the tower in the hash table is continued until:

Case 1. The *Worker* encounters a leaf in the search tree, i.e. a level that cannot be placed as an extension to the current solution. Suppose that the *Worker* must backtrack to level k. If $k < k_0$, then the *Worker* sends a message *DeepBack* to the *Master* along with the value of k. $k \geq k_0$ means that the *Worker* has placed some levels of the tower. A minimum number of levels which the *Worker* must place before communicating with the *Master* is a parameter to the algorithm. This parameter can be viewed as a *grain size* or *granularity* of search work. We denote it by s. If $k \geq k_0 + s$, i.e. the *Worker* has placed a required number of levels, then it sends to the *Master* the *U*-values for the placed levels. Otherwise ($k < k_0 + s$ holds) it continues the search.

Case 2. The *Worker* placed the last level of the tower in the hash table. In such a case the *Worker* sends to the *Master* the *U*-values for the placed levels and the information that the solution has been found.

To accelerate the initial phase of the search, all the *Workers* after placing the first level with $U[1] = 0$ continue the search until a backtrack occurs. Then, the first *Worker* sends to the *Master* a message containing the number of placed levels and the corresponding *U*-values. Each *Worker* backtracks then a number of levels equal to its number and resumes the search, i.e. the first *Worker* backtracks one level, the second *Worker* two levels etc. If a *Worker* is to backtrack below the second level, it frees all places in the hash table except for the places occupied by the words on the first level, and waits for a *New* message.

4.1.2. *Master* process

The *Master* holds partial solutions and assigns work to *Workers*. It stores pointers to last nodes processed by each *Worker* and keeps track of idle *Workers*. When active nodes become available, they are sent to idle *Workers*. Once a node is assigned to a *Worker* its assign flag is set. The *Master* receives three types of messages from *Workers*:

PartialSuccess. This message is sent when a *Worker* placed successfully some levels, and then encountered a level which cannot be placed. It contains the number of levels placed along with the corresponding *U*-values. Suppose that the *Worker* placed n levels beginning from level i, and on level $i+n$ it backtracked to level $i+n-1$. Let the *U*-values received be $U[i]$, $U[i+1]$, ..., $U[i+n-1]$. If the last node processed by the *Worker* was the first node on the *ActiveNodes* list, the Master sends the *Continue* message to the *Worker*. Then it changes the *U*-value of the last node processed by the *Worker* to $U[i]$, and inserts nodes $(i+1, U[i+1])$, $(i+2, U[i+2])$, ..., $(i+n-1, U[i+n-1])$ in the r-trie. The *Master* also adds to the r-trie new active nodes $(i, U[i]+1)$, $(i+1, U[i+1]+1)$, ..., $(i+n-2, U[i+n-2]+1)$. These are alternatives for expansion. If the *Worker* did not process the first node on the *ActiveNodes* list, such a node is found and its corresponding partial solutions is computed. This solution is compared with the one received from the *Worker*. The fragment in which they differ is sent back to the *Worker* as a *New* message.

DeepBack. This message is sent when a *Worker* executed backtrack pruning (see sec. 2). It contains the level number to which the *Worker* backtracked. Let this number be i, and let the last node processed by the *Worker* be $(j, U[j])$. The *Master* traverses the r-trie starting from node $(j, U[j])$ until node $(i, U[i])$ is encountered. Then it removes from the r-trie all not assigned active descendants of node $(i, U[i])$, and creates and sends a *New* message to the *Worker*.

TotalSuccess. This message is sent when a *Worker* placed successfully the last level of the tower. It contains the *U*-values for the levels placed.

Example 3. Consider an example search carried out by *Workers* P_1, P_2 and P_3, and the *Master*. Initially, the r-trie consists of two nodes: (1, 0) and (2, 0) (Fig. 2a). Suppose that during the search the following messages are sent to the *Master*:

Message no.	Message type	from *Worker*	*U*-values
1	*PartialSuccess*	P_1	(5, 2, 6)
2	*PartialSuccess*	P_1	(8, 1, 5)
3	*DeepBack*	P_2	to level 2
4	*PartialSuccess*	P_3	(7, 5)

In response to message 1 the *Master* changes the *U*-value of node (2, 0) to 5 and inserts in the r-trie new nodes (3, 2), (4, 6), (2, 6) and (3, 3). Nodes (4, 6), (3, 3) and (2, 6) become active (Fig. 2b). As mentioned earlier, P_1 backtracks one level so it continues the search from node (4, 6);. P_2 continues from node (3, 3), and P_3 from node (2, 6). Suppose that P_1 has placed successfully the next three levels (message 2). The *Master* sends to P_1 a *Continue* message since the node (4, 6) is the first node on the *ActiveNodes* list. Then it changes the U-value of node (4, 6) to 8 and inserts nodes (5, 1), (6, 5), (4, 9) and (5, 2) as active nodes in the r-trie. Now suppose that P_2 sends message 3. Having received it the *Master* removes from the r-trie node (3, 3) and sends to P_2 a new partial solution (by making use of a *New* message) beginning from level 3 with U-values (2, 8, 2). Thus, P_2 will continue the search from level 5. Suppose that P_3 has placed successfully the next two levels (message 4). The *Master* changes the U-value of node (2, 6) to 7 and inserts in the r-trie nodes (3, 5) and (2, 8). Now the first active node not assigned yet is (4, 9). The *Master* sends to P_3 a new partial solution beginning at level 2 with U-values (5, 2, 9). Thus P_3 will continue the search from level 4. Fig. 3 shows a state of the r-trie at that moment. Active nodes (6, 5), (5, 2) and (4, 9) are assigned. □

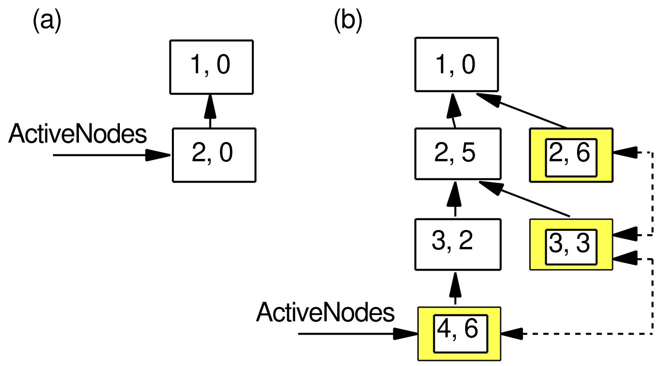

Figure 2: (a) Initial state of the r-trie; (b) The r-trie after message 1 has been processed

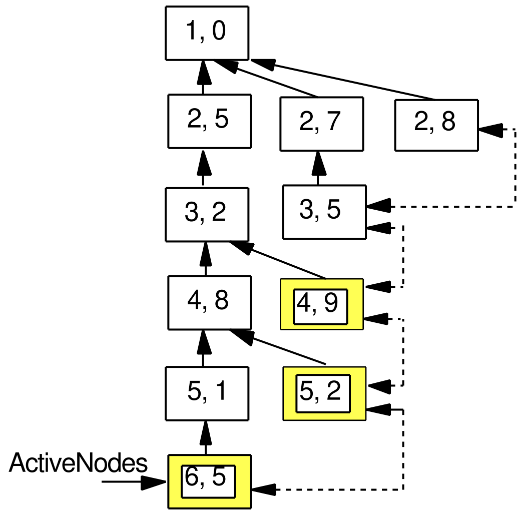

Figure 3: The r-trie after message 4 has been processed

4.2. Occam implementation

Here we discuss the major differences between the occam implementation of the algorithm and the Parallel-C implementation presented above.

4.2.1. *Worker* processes

A *Worker* process receives one extra message from the *Master*, the *End* message. This has no components and is sent when the solution has been found.

This implementation has no backtrack pruning, thus with the *DeepBack* message a *Worker* sends no k value.

4.2.2. *Master* process

The *Master* process receives only two types of messages from the *Worker* processes:

Success. This is exactly the same message as the *PartialSuccess* message. When the *Master* receives the *Success* message with the last level placed, it sends a single *End* message to the *Workers*.

DeepBack. This message is sent when a *Worker* backtracks one level below the start level. It has no components, as mentioned above.

4.2.3. The farming harness

In the occam implementation a farming harness was written specifically for the application. Such a harness consists of two processes:

Splitter. This process performs the distributon of jobs from the *Master* to the farm. As the jobs are numbered for specific *Workers*, the *Splitters* direct each one accordingly.

Merger. This process collects results from the *Workers* and passes them back to the *Master*. In times of choice the *Mergers* give priority to the results from the *Worker* not the neighbouring *Merger*.

These processes run at high priority so that messages (jobs and results) destined for other processors can be passed on immediately.

Fig. 4 shows a farm of three transputers.

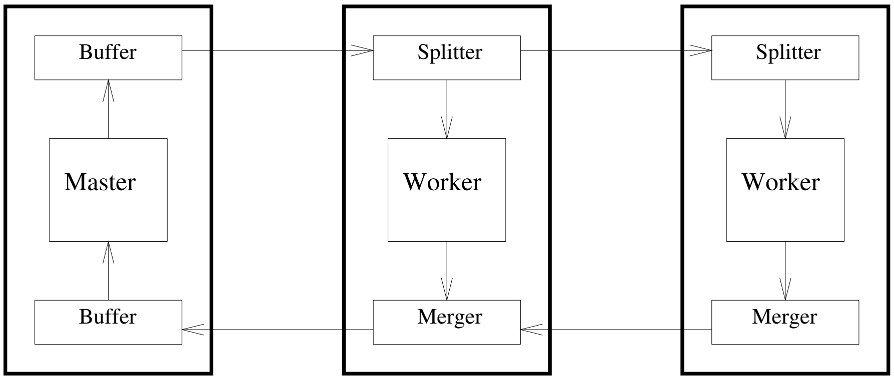

Figure 4: An example farm

5. EXPERIMENTAL RESULTS

The parallel algorithm was implemented in Parallel-C under control of the Express Parallel Programming Environment and in occam on UKC's Meiko Surface. We ran the algorithm on a transputer system with ten T800 20 MHz transputers configured into a linear array. The *Master* was run on the first transputer, and the *Workers* were run on the remaining transputers. The experiments were conducted for the six sets of words of sizes $m = 50, 60, \ldots, 100$. For these sets tables 1 and 2 show the execution times of the sequential search (using only one transputer) in Parallel-C and occam. Fig. 5 and fig. 6 show the speedup of the Parallel-C and occam implementations respectively, as a function of the number of processors. The granularity of search work, s, is a parameter to the graphs. Each point of a graph was computed as an average over the 36 results measured for all the sets and the values of parameter $\beta = 0.45, 0.46, \ldots, 0.5$.

m	$\beta = 0.45$	$\beta = 0.46$	$\beta = 0.47$	$\beta = 0.48$	$\beta = 0.49$	$\beta = 0.5$
50	180.818	85.126	63.051	25.91	8.893	3.239
60	183.126	116.579	48.067	19.989	5.82	3.188
70	218.984	128.58	99.849	51.777	33.204	6.387
80	312.699	83.06	56.375	25.222	11.843	5.492
90	174.454	126.423	76.583	51.919	13.223	4.778
100	329.219	201.531	63.777	39.217	13.407	4.052

Table 1: Sequential search times in seconds (Parallel-C), $|g| = \beta m$

m	$\beta = 0.45$	$\beta = 0.46$	$\beta = 0.47$	$\beta = 0.48$	$\beta = 0.49$	$\beta = 0.5$
50	227.658	195.883	112.862	84.468	58.293	21.217
60	382.592	260.768	80.682	73.845	44.657	10.769
70	385.010	250.005	144.504	66.800	57.800	13.147
80	318.291	200.772	205.961	45.931	29.858	17.782
90	402.634	322.956	181.241	123.641	55.215	14.851
100	865.847	445.430	244.551	115.510	57.919	44.709

Table 2: Sequential search times in seconds (occam), $|g| = \beta m$

6. CONCLUSIONS

We presented the parallel algorithm for conducting a search for a first solution to the problem of generating MPHFs. The algorithm uses the technique of the processor farm, and adopts the idea of delayed-release [4]. The special data structure, called reversed trie (r-trie), was devised to carry out the search. The experiments showed that the parallel algorithm exhibits a consistent and almost linear speedup (cf. Figs. 5 and 6). As a grain size of the search work increases, a superlinear speedup can be obtained (cf. Fig. 5). The r-trie structure proved to be highly memory efficient.

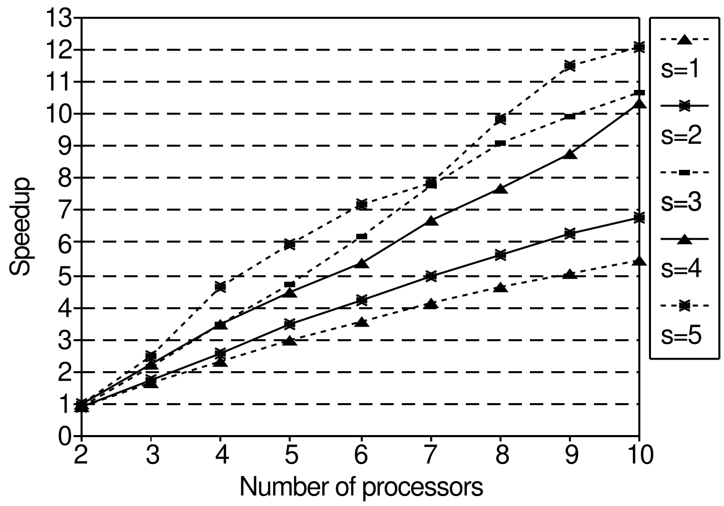

Figure 5: Speedup versus number of processors (Parallel-C)

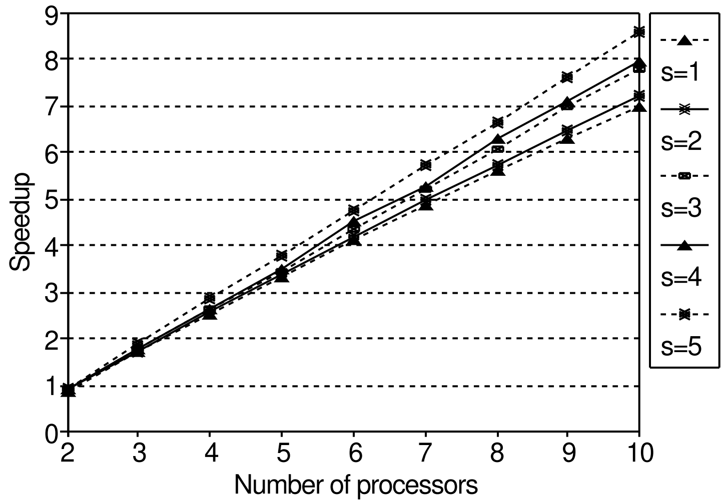

Figure 6: Speedup versus number of processors (occam)

REFERENCES

[1] Z.J. Czech and B.S. Majewski, A linear time algorithm for finding minimal perfect hash functions, *The Computer Journal* **36** (4) (1993).

[2] Z.J. Czech and B.S. Majewski, Generating a minimal perfect hash function in $O(M^2)$ time, *Archiwum Inform. Teoret. i Stos.* **4** (1-4) (1992) 3-20.

[3] T.H. Lai and S. Sahni, Anomalies in parallel branch-and-bound algorithms, *Comm. ACM* **27** (6) (1984) 594-602.

[4] L.V. Kalé and A. Saletore, Parallel state-space search for a first solution with consistent linear speedups, *Intern. Journ. of Parallel Program.* **19** (4) (1987) 251-293.

[5] D.E. Knuth, *The Art of Computer Programming, Vol. 3, Sorting and Searching* (Addison-Wesley, New York, 1973) 481-499.