[KLP89] Z. Kedem, G. Landau, and K. Palem. Optimal parallel suffix-prefix matching algorithm and applications. *Proc. 1st Annual ACM Symposium on Parallel Algorithms and Architecture*, 1989, 388-398.

[KMP77] D.E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1973), 323-350.

[KMR72] R. Karp, R. Miller, and A. Rosenberg. Rapid Identification of Repeated Patterns in Strings, Trees and Arrays. *Proc. 4th Annual ACM Symposium on Theory of Computation*, 1972, 125-136.

[KP88] Z. Kedem and K. Palem. Optimal parallel algorithms for forest and term matching. *Theoretical Computer Science*, 93, 1992, 245-264.

[O83] M. Overmars. The design of dynamic data structures. *Lecture Notes in Computer Science*, Springer-Verlag, 156, 1983.

[PVW83] W. J. Paul, U. Vishkin, and H. Wagener. Parallel dictionaries on 2—3 trees. *Proc. ICALP*, 1983, 597-609.

[Rab93] M. Rabin. Optimal parallel pattern matching through randomization. *Sequences '91: Methods in Communication, Security and Computer Science*, R. Capocelli, A. De Santis and U. Vaccaro Ed., Springer-Verlag 1993.

[Rag90] P. Ragde. The parallel simplicity of compaction and chaining. *Proc. of the 17th ICALP, Springer LNCS 443*, 744-751, 1990.

[Vi85] U. Vishkin. Optimal pattern matching in strings. *Information and Control*, Vol. 67, 1985, 91-113.

[Vi90] U. Vishkin. Deterministic sampling—A new technique for fast pattern matching. In *Proc. of the Twenty Second Annual ACM Symposium on Theory of Computing*, 1990, 170-180.

[Br74] R. Brent. The parallel evaluation of general arithmetic expressions. *Journal of ACM*, 1974, 21:202-206.

[CGRMR92] M. Crochemore, L. Gasieniec, W. Rytter, S. Muthukrishnan, and H. Ramesh. Fast parallel two dimensional/string pattern matching. *Manuscript*, 1992.

[CV89] R. Cole and U. Vishkin. Faster optimal parallel prefix sums and list ranking. *Information and Computation*, 1989, 81, 334-352.

[F93] M. Farach. *Personal Communication.*

[FL70] M. Fischer and L. Ladner. Parallel prefix computations. *Journal of ACM*, Vol. 27, No. 4, 1980, 831-838.

[Ga84] Z. Galil. Optimal Parallel Algorithms for String Matching. *Proc. ACM Symposium on Theory of Computation*, 1984. Also in *Information and Control*, Vol. 67, 144-157, 1985.

[Ga92] Z. Galil. Hunting lions in the desert optimally or a constant time optimal parallel string matching algorithm. *Proc. of the 23rd ACM Symposium on Theory of Computation*, 1992.

[Gi93] R. Giancarlo. The suffix trees of a square matrix with applications. *Proc. ACM Symposium on Discrete Algorithms*, 1993, 402-411.

[GG93] R. Grossi and R. Giancarlo. Parallel construction of suffix trees of matrices. To appear in the *Proc. of 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[GMV91] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. *Proc. IEEE Foundations of Computer Science*, 698-710, 1991.

[H88] T. Hagerup. On saving space in parallel computation. *Information Processing Letters*, Vol. 29, 1988, 327-329.

[H92] T. Hagerup. The log-star revolution. *Proc. of 9th Annual Symposium on Theoretical Aspects of Computer Science*, Springer Lecture Notes in Computer Science, Vol. 577, 1992, 259-278.

[H93] T. Hagerup. Fast deterministic processor allocation. *Proc. of the 4th Annual ACM Symposium on Discrete Algorithms*, 1993, 1-10.

[IS91] R. Idury and A. A. Schaffer. Dynamic dictionary matching with failure functions. *Proc. Third Symp on Combinatorial Pattern Matching*, 1991, 273-284.

[Ja92] J. JáJá. *An introduction to parallel algorithms.* Addison-Wesley Publ., 1992.

Bibliography

[AB92] A. Amir and G. Benson. Two-dimensional periodicity in rectangular arrays. *Proc of the 3rd ACM Symposium on Discrete Algorithms*, 1992, 440-452.

[ABF93] A. Amir, G. Benson, and M. Farach. Parallel two dimensional matching in logarithmic time. To appear in *Proc. 5th ACM Symposium on Parallel Algorithms and Architectures*, 1993.

[AC75] A.V. Aho and M.J. Corasick. Efficient string matching: An aid to bibliographic search. *Communications of ACM*, 18(6), 333-340.

[AF91] A. Amir and M. Farach. Adaptive dictionary matching. *Proc 32nd IEEE Symp Foundations of Computer Science*, 1991, 760-766.

[AF92] A. Amir and M. Farach. Two dimensional dictionary matching. *Information Processing Letters*, Vol. 44, 1992, 233-239.

[AFGGP91] A. Amir, M. Farach, R. Giancarlo, Z. Galil, and K. Park. Dynamic dictionary matching. *Manuscript*, 1991.

[AFILS93] A. Amir, M. Farach, R. Idury, J. La Poutre', and A. Schaffer. Improved dynamic dictionary matching. *Proc. 4th Annual ACM Symposium on Discrete Algorithms*, 1993, 392-401.

[AFM92] A. Amir, M. Farach, and Y. Matias. Efficient randomized dictionary matching algorithms. *Proc. 3rd Symp on Combinatorial Pattern Matching*, 1992, 259-272.

[AILSV88] A. Apostolico, C. Iliopoulos, G. Landau, B. Schieber, and U. Vishkin. Parallel Construction of a Suffix Tree with Applications. *Algorithmica* 3, 1988, 347-365.

[B77] R. Bird. Two Dimensional Pattern Matching. *Information Processing Letters* 6:168-170, 1977.

[Ba78] T. Baker. A Technique for Extending Rapid Exact String Matching to Arrays of More Than One Dimension. *SIAM J. Computing* 7:533-541, 1978.

[BDHPRS91] P.C.P. Bhatt, K. Diks,T. Hagerup, V.C. Prasad, T. Raznik, and S. Saxena. Improved deterministic parallel integer sorting. *Information and Computation*, 94, 1991, 29-47.

[BG90] D. Breslauer and Z. Galil. An optimal $O(\log\log m)$ time parallel string matching algorithm. *SIAM J. Comput.*, 19(1990), pp. 1051-1058. Also in, O. Berkman, D. Breslauer, Z. Galil, B. Scheiber, and U. Vishkin. Highly Parallelizable Problems. In *Proc. 21st ACM Symp. on Theory of Computing*, 1989.

length, same as in Step $3a$, denoted by $\mathcal{R}$. As in Step $3a$, corresponding to each $P_j \in \mathcal{D}$, generate tuples of the form $\langle\langle P_j(1), \delta(P_j^s), \delta'(r(P_j^s))\rangle, \beta(P_j)\ \rangle$. Let this be set S_p. Form set S_t by generating, for all even i in each T_j,

$$\langle\ T_j(i), \alpha(T_j(i+1)), \delta'(T_j(i+|\alpha(T_j(i)|+1)\cdots T_j(i+|\alpha(T_j(i)|+\mathcal{R}))\rangle.$$

As in Step $3b$, namestamp set S_t with S_p. The stamp for the tuple corresponding to $T_j(i)$ gives the name of the pattern from $\mathcal{D}$ that matches T_j beginning at i where i is even.

That completes the description of our algorithm. Step 1 takes $O(1)$ time and $O(n+M)$ work. Let $OT(n, M, m)$ and $OW(n, M, m)$ represent the time and work complexity of our algorithm given text size n, dictionary size M such that each pattern is of length m. Step 2 takes $OT(n/2, M/2, m/4)$ time and $OW(n/2, M/2, m/4)$ work. Step $3a$ takes $O(1)$ time and $O(M)$ work. Steps $3b$ and $3c$ each take $O(1)$ time and $O(n+M)$ work. Hence,

$$OT(n, M, m) \leq OT(n/2, M/2, m/4) + 1.$$
$$OW(n, M, m) \leq OW(n/2, M/2, m/4) + n + M.$$

Theorem 11 *Given a text of size n, a dictionary of size M consisting of patterns each of size m, static dictionary matching can be performed in $O(\log m)$ time and $O(n+M)$ work.*

This result also yields optimal algorithms for multi-dimensional pattern matching and other problems such as *suffix-prefix* matching studied in [KLP89,Rab93]. Rabin [Rab93] provides randomized algorithms for these problems. Our algorithms, as those in [KLP89] are deterministic and are much simpler and as efficient as the ones presented there.

Acknowledgements: We sincerely thank Martin Farach for several fruitful discussions concerning [AF91,AFM92]. We are also grateful to Amihood Amir, Martin Farach, Zvi Galil, Raffaelle Giancarlo, and Rajeev Raman for discussions related to this work.

T'. This returns a name for each string $p \in P'$ denoted by $\delta(p)$. In addition, for each location i in each $T'_j \in T'$, this returns the name of the pattern in P' that matches beginning at that location. Equivalently, for each odd text location $T_j(i) \in T$, the name of the pattern from P' which matches at that location has been determined. Let this be denoted by $\alpha(T_j(i))$.

Recall that in Section 4.1, prefix names were computed for $\mathcal{D}$. At each level in recursion, the prefix names for a new set of patterns for lower levels of recursion could be gleaned from those of $\mathcal{D}$ because the new set of patterns were prefixes of $\mathcal{D}$. In our recursive step here, not all strings in P' are prefixes of $\mathcal{D}$. Hence, names for strings in P' can not be directly derived from those of $\mathcal{D}$. One approach towards generating the names for P' is performing a prefix-naming computation at each stage of recursion. However, each stage would then take $O \log m)$ time leading to an overall algorithm that works in $O(\log^2 m)$ time. Our approach here has been to embed the computation of the names for the pattern strings within the recursive framework by strengthening the recursive invariant.

Step 3. In this step, we perform extension to the right and extension to the left as in Section 4.1 and Section 4.4. In addition, we generate the prefix-names for the strings in the set $\mathcal{D}$ so the recursive invariant is maintained.

Step 3a. We find the names for the strings in $\mathcal{D}$. Consider each string $P_j \in \mathcal{D}$. Let the string obtained from P_j^p by shrinking by 4 be P'_j. The name $\delta(P'_j)$ is known. From $\delta(P_j^p)$, we compute $\beta(P_j)$, the name for $P_j \in \mathcal{D}$. Let $r(P_j^p)$ denote the residue when P_j^p is shrunk by a factor of 4. The residue is of length at most 3. The residue for each pattern is of same length denoted by $\mathcal{R}$. Assume that all substrings in T and $\mathcal{D}$ of length $\mathcal{R}$ have been named using the function δ'. Now, for each $P_j \in \mathcal{D}$, generate the tuple $\langle \delta(P_j^p), \delta'(r(P_j^p)), P_j(|P_j|) \rangle$. Name this set of tuples. The result gives the name $\beta(P_j)$ for $P_j \in \mathcal{D}$.

Step 3b. For each odd position in the text, we find the index of the pattern that matches beginning at that position. As in Step 3a, corresponding to each $P_j \in \mathcal{D}$, generate a tuple of the form $\langle \langle \delta(P_j^p), \delta'(r(P_j^p)), P_j(|P_j|) \rangle, \beta(P_j) \rangle$. Let this be set S_p. Form set S_t by generating, for all odd i in each T_j,

$$\langle \alpha(T_j(i)),\ \delta'(T_j(i + |\alpha(T_j(i)| + 1) \cdots T_j(i + |\alpha(T_j(i)| + \mathcal{R})), T_j(i + |\alpha(T_j(i)| + \mathcal{R} + 1) \rangle.$$

Namestamp S_t with S_p. The stamp for the tuple corresponding to $T_j(i)$ gives the name of the pattern from $\mathcal{D}$ that matches T_j beginning at i where i is odd.

Step 3c. (Extend-Left) For each even position in the text, we determine the pattern that matches beginning at that position, by extending the pattern from P' that matches at the neighbor to the right. For each $P_j \in \mathcal{D}$, $\delta(P_j^s)$ denotes the name of the string in P' generated from P_j by dropping the leading symbol and shrinking the resultant string by a factor of 4. Let the residue of P_j^s be denoted by $r(P_j^s)$. All such residues are of equal

Theorem 10 *The dictionary $\mathcal{D}_0$ is processed in $O(\log M_0)$ time and $O(M_0 \log M_0)$ work. The insert($\mathcal{P}, \mathcal{D}_{i-1}$) operation takes $O(\log M_{i-1})$ amortized time and $O(\mu_i \log M_{i-1})$ amortized work. The operation delete($\mathcal{P}, \mathcal{D}_{k-1}$) can be performed in $O(\log M_{k-1})$ amortized time and $O(\eta_k \log M_{k-1})$ amortized work. The operation match $(t_j, \mathcal{D}_{i'})$ takes $O(\log M_{i'})$ time and $O(n_j \log M_{i'})$ work.*

7 Multi-pattern String Matching and Related Problems

We modify the algorithm for static dictionary matching in Section 4 to derive simple optimal algorithms when all the patterns in the dictionary are of the same length; this is the *multi-pattern string matching* problem of [KLP89]. In this process we make some basic observations which we believe are critical in using the shrink-and-spawn technique to derive optimal speed-up algorithms, even when the overall work is linear in the input size. The main idea is similar in spirit to that in Section 4.4 where we discussed issues in shrinking the text sizes by a fraction recursively. When all patterns are of the same length, this step and the corresponding step of extending the match to the left when returning from lower levels of recursion can be achieved efficiently. The details are as follows.

Let T be the set of text strings and let $\mathcal{D}$ denote the set of pattern strings each of length m. The total size of $\mathcal{D}$ is M. Unlike in Section 4 and other earlier sections where we first solve the prefix-matching version of the problem, in this section we solve the dictionary matching problem directly. Also as it turns out, we do not need the prefix-names for the strings in $\mathcal{D}$. The *name* of a pattern string, defined to be the prefix-name of the last location in it, is sufficient. For reasons to be specified later, we maintain a stronger recursive invariant of generating the names at each recursive level, for the appropriate set of strings derived from the dictionary at that level; recall that in Section 4, the prefix-names were computed in one step that preceded the recursive prefix-matching step. Our algorithm proceeds as follows:

Step 1 (Optimal Shrink-and-spawn) Let $\mathcal{P}$ be obtained from $\mathcal{D}$ by replacing each $P_j \in \mathcal{D}$ by the following two strings: P_j^s, a suffix obtained by dropping the leading symbol in P_j and P_j^p, a prefix obtained by dropping the last symbol in P_j. All strings in $\mathcal{P}$ are of same length. Let δ' be a naming function for all the substrings of length 4 in T and $\mathcal{P}$. Shrink each string in $\mathcal{P}$ by 4 to get $\mathcal{P}'$. The residues of length at most 3 would be considered later. Spawn 4 copies of each string in T. Note that for a string $t \in T$, the four copies are t^i for $1 \leq i \leq 4$ (See the definition of shrink-and-spawn from Section 3). Delete alternate strings t^2 and t^4 for each $t \in T$. Let the resultant set be T'. Note that the factor by which the text is spawned is half the factor by which the dictionary strings are shrunk. As a result, $\mathcal{P}'$ is of size $2 \times M/4 = M/2$ and T' is of size $n/2$.

Step 2. (Recursive Step) Recursively, solve the dictionary matching problem on P' and as providing optimal algorithms for integer sorting. Note that no deterministic optimal algorithms are known for the integer-sort problem.

Following this claim, dynamic stamp-counting can be done deterministically within following bounds: 1. processing S_0 in $O(\log|S_0|/\log\log|S_0|)$ time and $O(|S_0|\log\log|S_0|)$ work [BDHPRS91]. 2. *insert*(s, S_i) and *delete*(s, S_i) in $O(1)$ time and $O(1)$ work. 3. *namestamp*(A, S_i) in $O(1)$ time and $O(|A|)$ work. Dynamic stamp-listing can be performed using integer-sorting using space quadratic in M when the sets are tuples with two elements, each in the range $1 \cdots M$, as follows. Essentially keep a double linked list of the tuples with a particular stamp on top of an array of size M for each stamp. In all our applications of dynamic stamp-listing, the tuples have only two elements; hence, the space utilized is at most quadratic in the dictionary size. It follows that the abovementioned bounds are achievable for dynamic stamp-listing as well.

We now return to fully dynamic prefix-matching. As in partly dynamic prefix-matching, we simulate parts of the static prefix-matching algorithm. Again the various standard namestamping steps are replaced by the appropriate variants as follows: (i) namestamping in prefix-naming and naming (shrink-and-spawn step) are replaced by partly dynamic namestamping, (ii) namestamping in Extend-Right is replaced by dynamic stamp-counting, and (iii) namestamping in the retrieve-index problem (Phase 2) is replaced by stamp-listing.

Theorem 9 *The initial dictionary $\mathcal{D}_0$ can be processed in $O(\log M_0/\log\log M_0)$ time and $O(M_0\log\log M_0)$ work deterministically. The insert$(\mathcal{P}, \mathcal{D}_i)$ operation can be performed in $O(\log\mu_i)$ time and $O(\mu_i)$ work. The delete$(\mathcal{P}, \mathcal{D}_{k-1})$ operation can be performed in $O(\log M_k/\log\log M_k)$ time and $O(\eta_k\log\log M_i)$ amortized work. A match$(t_j, \mathcal{D}_{i'})$ is performed in $O(\log m_{i'})$ time and $O(n_j\log m_{i'})$ work.*

Consider inserting or deleting several pattern strings simultaneously. The following modifications are incorporated. The various tuples which are simultaneously inserted or deleted are sorted using the integer-sorting algorithm. In dynamic name-listing, the list of tuples with the same stamp are kept in a 2—3 tree which can be updated as in [PVW83]. Dynamic stamp-counting and stamp-listing with these modifications yield dynamic prefix-matching algorithms within bounds slightly worse than those cited in Theorem 9.

6.2.2 Fully Dynamic Dictionary Matching

Combining our result for the fully dynamic prefix matching problem with techniques of Amir, Farach and Matias [AFM92,F93], an algorithm for fully dynamic dictionary matching is derived. As in partly dynamic dictionary matching, a trie of pattern strings is maintained. In addition, UNION/FIND operations are implemented on the set of marked nodes in the trie. This keeps track of the available marked ancestors for each marked node as patterns get deleted. To sum, using a fast prefix-sum computation [CV89] and the various tables are rebuilt. Hence, the work bounds for deletions are amortized.[6] Again we first consider *fully dynamic prefix-matching* in which the output for the operation $match(t_j, \mathcal{D}_{i'})$ is for each text location $t_j(k)$, the longest prefix from $\mathcal{D}_{i'}$ which occurs beginning at $t_j(k)$.

6.2.1 Fully Dynamic Prefix-Matching

We describe modifications to our algorithm in Section 4.1. Our modifications concern namestamping that yield more sophisticated variations to be used in the fully dynamic case. This is because, deleting patterns brings up issues not encountered earlier. Recall from Section 3 that to namestamp with S_1, each tuple $\langle x, y\rangle$ in S_1 is assigned a namestamp $l(x)$. When a tuple with element x is deleted, we need to change $l(x)$. Two possibilities occur: if deleting x leaves no tuples in S_1 with element x, $l(x)$ has to be cleared. On the other hand, if deleting x leaves some tuples in S_1 with element x, no changes need be made provided we are not particular about the namestamp. To ensure this, we need to only keep track of the number of tuples that have the same element. To satisfy this requirement, we define below a variation of namestamping called *dynamic stamp-counting.* Assuming now that we are particular about the namestamp $l(x)$, it is not sufficient to keep track of the number of tuples with an element x; we need to additionally keep track of the namestamps of these tuples. Corresponding to this, we define a variant of namestamping called *dynamic stamp-listing.*

Now we formally define two variants of namestamping. Consider *dynamic stamp-counting.* Initially we are given a set of tuples S_0. To each distinct element x in S_0, we assign namestamp $l(x)$ as in standard namestamping. In addition, we assign to x a *count* $c(x)$ of the number of tuples in S_0 with element x. The following sequence of operations is performed: (i) $insert(s, S_i)$, adds a tuple s to S_i and $S_i \bigcup S = S_{i+1}$ as defined in partly dynamic namestamping. (ii) $delete(s, S_i)$, removes a tuple s from S_i and $S_i - S = S_{i+1}$. (iii) $namestamp(A, S_i)$, i.e., namestamp set A as in standard namestamping. *Dynamic stamp-listing* is defined analogous to dynamic stamp-counting by replacing $c(x)$ with $p(x)$ which is a pointer to a *list* of all tuples $\langle x_1, y_1\rangle \in S_i$ such that $x = x_1$.

In our applications, all the integers in dynamic namestamping would be in the range $[1 \cdots M]$ where M is the dictionary size. We claim that dynamic stamp-counting is exactly as hard as the *integer-sort problem* [BDHPRS91], that is, sorting M numbers in the range $[1 \cdots M]$. Obviously if the elements of S_0 are sorted, the namestamps and counts can be assigned in $\alpha(M)$ time and $O(M)$ work using the standard algorithm to find nearest 1, for each 0, in a boolean array [Rag90]. Also, if stampcounting the set S_0 is accomplished, the set of tuples in S_0 can be sorted using prefix-sum computations which take $O(\frac{\log M}{\log\log M})$ time and $O(M)$ work [CV89]. Hence, stamp-counting is at least as hard

[6] We use a simple amortization here. Assume a sequence of operations the ith of which involves w_i work. After a sequence such that $\sum w_i = cM$ for a constant fraction c, we have an operation that takes $f(M)$ work for some function f. The amortized work for the ith operation in this case is defined as $(w_i/cM) \times f(M)$.

Our description above carries over to the case when several pattern strings are inserted simultaneously.

6.1.2 Partly Dynamic Dictionary Matching

Following [AFM92,AF91,F93], a parallel algorithm for the following problem is easily derived: given a prefix from the dictionary that is dynamic under *insert* operations, determine its longest prefix that is a pattern. Essentially, the following simple subset of the techniques developed in [AFM92] is sufficient. Maintain a trie of the pattern strings. Initially, the suffix tree of the dictionary can be modified to serve as the trie. Certain nodes in the trie are marked corresponding to the patterns in the dictionary. The query concerning the longest prefix of a given prefix that is a pattern in the dictionary, translates to determining the nearest marked ancestor in this trie. To maintain this information dynamically, they maintain the Euler tour of this trie in a balanced tree.

Inserting a new pattern is performed as follows. From the description of the algorithm for partly dynamic prefix-matching, it follows that the longest prefix of the new pattern that is present in the current dictionary is available. This information can readily be obtained as a pointer to the node in the trie with the corresponding prefix. At this node, the new pattern is inserted. In [AFM92], it is described how to maintain the Euler tour when patterns are inserted. Given this algorithm, our result from Theorem 7 immediately yields,

Theorem 8 *Processing $\mathcal{D}_0$ takes $O(\log M_0)$ time and $O(M_0 \log M_0)$ work. The operation insert $(\mathcal{P}, \mathcal{D}_{i-1})$ takes $O(\log M_{i-1})$ time and $O(\mu_i \log M_{i-1})$ work. The operation match$(t_j, \mathcal{D}_{i'})$ takes $O(\log M_{i'})$ time and $O(n_j \log M_{i'})$ work. The output for each text location is the longest pattern that matches beginning at that location.*

Once again, a detail concerns dynamic space allocation for maintaining the tries. As mentioned in the Section 6.1.1, an amortized solution is immediate; this can be made into a worst case solution using standard methods [O83].

6.2 Fully Dynamic Dictionary Matching

The *fully dynamic dictionary matching* problem admits the operation of deleting patterns from the dictionary in addition to those supported by the partly dynamic dictionary matching problem. Let the kth operation that modifies the dictionary (through insertions and deletions) be *delete*$(\mathcal{P}, \mathcal{D}_{k-1})$ where $\mathcal{P}$ has length η_k. This operation involves removing the pattern $\mathcal{P}$ from $\mathcal{D}_{k-1}$ resulting in dictionary $\mathcal{D}_k$. We modify our static dictionary matching algorithm from Section 4.

When a *delete* operation is performed, the pattern to be deleted is not removed from the dictionary. It is simply "marked". When the total size of the patterns in the dictionary falls below a fraction of the size of the tables, the marked patterns are squeezed out in prefix-naming, naming (shrink-and-spawn), Extend-Right and retrieve-index problems (Phase 2).

Theorem 7 *The initial dictionary $\mathcal{D}_0$ in which the longest pattern is of length at most m_0 can be processed in $O(\log m_0)$ time and $O(M_0)$ work. Following this, the insert($\mathcal{P}, \mathcal{D}_i$) operation can be performed in $O(\log \mu_i)$ time and $O(\mu_i)$ work. A match($t_j, \mathcal{D}_{i'}$) is performed in $O(\log m_{i'})$ time and $O(n_j \log m_{i'})$ work. For each location $t_j(k)$, the output is the longest prefix from $\mathcal{D}_{i'}$ that matches beginning at $t_j(k)$.*

A detail concerns dynamic space allocation. Note that as patterns get inserted, the size of the arrays needed to perform namestamping increases. Such a situation arises in the parallel construction of 2—3 trees [PVW83] in which it is assumed that an array of infinite size is provided. The 2—3 tree is maintained at the beginning of this array dynamically. However, we feel that this is not a realistic assumption, especially in our case where we are concerned the availability of several two dimensional arrays. We provide the following solution for this problem.

An amortized solution is immediate: given the initial dictionary size M_0, procure tables of size $2M_0 \times 2M_0$ for namestamping. Now, patterns of total size M_0 can be added to this initial dictionary using the algorithm outlined above. When the size of the inserted patterns goes beyond M_0, fresh tables of size $4M_0 \times 4M_0$ are procured. The old tables are copied into this larger table (takes $O(M_0)$ work corresponding to $O(M_0)$ total work done so far constructing this table). Now, the algorithm proceeds as earlier with the new tables. The old tables are discarded. Various complexity bounds stated in the previous theorem remain unchanged except that the bounds for insertions become amortized bounds. An important detail concerns copying the old tables into new tables; copying is done by simulating the dictionary processing algorithm, on the new table with old dictionary, rather than entry-by-entry copy of the tables. Note that this is particularly critical since copying the tables entry by entry would result in quadratic work.

This amortized bound can be made worst case bound using a variation of standard techniques for dynamizing data structures [O83]. Essentially, the technique is the following: assume that tables of size $2M_0 \times 2M_0$ have been used for processing the dictionary initially. Dynamic insert operations are performed till patterns of total size M_0 are inserted into the dictionary. Following this, tables of size $4M_0 \times 4M_0$ are procured. Without copying the old tables into the new tables, the algorithm continues working on the new table. When any insert is processed, two operations are performed, namely, inserting into the new table as dictated by the algorithm being careful to read any relevant entries in the old table and copying portions of the old table into the new table. Again copying is done as in the amortized case, i.e., by simulating the dictionary processing algorithm. When new patterns of total size M_0 are inserted, the old tables would have been fully copied onto the new tables and therefore, the old tables can be discarded. Hence, the previous theorem holds with worst case bounds.

Furthermore, it can be matched against a set of text arrays of total size n in $O(\log m)$ time and $O(n \log m)$ work.

6 Dynamic Dictionary Matching

6.1 Partly Dynamic Dictionary Matching

In partly dynamic dictionary matching, we have an *initial* dictionary $\mathcal{D}_0$ and are given an arbitrary sequence of *insert* and *match operations*, on-line. The ith insert operation is *insert*$(\mathcal{P}, \mathcal{D}_{i-1})$ where pattern $\mathcal{P}$ has length μ_i. It involves adding $\mathcal{P}$ to the dictionary $\mathcal{D}_{i-1}$. The resulting dictionary is $\mathcal{D}_i$. The jth match operation is *match*$(t_j, \mathcal{D}_{i'})$ which involves matching the text t_j of n_j characters, into $\mathcal{D}_{i'}$ where there are exactly i' inserts in the sequence of operations before the jth match. The output for each location k in t_j, is the longest pattern from $\mathcal{D}_{i'}$ that matches at k. Each dictionary $\mathcal{D}_i$ has size M_i with a longest pattern of length m_i.

6.1.1 Partly Dynamic Prefix-Matching

First we consider the prefix-matching version of the problem. In *partly dynamic prefix-matching*, the output for the operation *match*$(t_j, \mathcal{D}_{i'})$ is for each text location $t_j(k)$, the longest prefix from $\mathcal{D}_{i'}$ which occurs beginning at $t_j(k)$. We modify our algorithm in Section 4.1 to perform partly dynamic prefix-matching.

Our modifications to the algorithm in Section 4.1 concern namestamping. We first define the *partly dynamic namestamping* problem; namestamping from Section 3 is henceforth called *standard namestamping.* A set of tuples S_0 is initially processed as in standard namestamping. A sequence of two operations can be performed. The operation *insert*(S, S_i) is the $i+1$st insertion in which a set S of tuples is added to S_i as follows: the elements in S are assigned namestamps *consistent* with those in S_i. That is, if an element x in S is in S_i, then $l(x)$ is unchanged. The elements in S and not in S_i are assigned namestamps as defined in Section 3. Following this operation, $S \bigcup S_i = S_{i+1}$. The operation *namestamp*(A, S_i) is performed after the ith insertion and before the $i+1$st, and it namestamps A with S_i as in standard namestamping. We claim that by modifying the standard namestamping procedure in Section 3 slightly, the following bounds are achieved for partly dynamic namestamping: S_0 is processed in $O(1)$ time using $O(|S_0|)$ processors following which *insert*(S, S_i) takes $O(1)$ time using $|S|$ processors and *namestamp*(A, S_i) takes $O(1)$ time using $O(|A|)$ processors.

We now return to partly dynamic prefix matching. The initial dictionary is preprocessed as in the static prefix-matching algorithm in Section 4.1 and the *insert*$(\mathcal{P}, \mathcal{D}_{i-1})$ operation is performed by simulating this algorithm for dictionary processing on $\mathcal{P}$. In both these steps, all standard namestamping procedures are replaced by partly dynamic namestamping. Recall that standard namestamping steps were involved in Section 4.1

Generate the set S_t of tuples from the text. The set $S_t(i)$ consists of tuples generated by those text locations τ for which $\alpha(\tau)$ is a $2i \times 2i$ square. Consider one such text location $\tau = T(j,k)$. Let its neighbor immediately to the right be τ^c and the neighbor immediately below be τ^r. Let the longest square-prefixes of P' that match at τ^r and τ^c be $\alpha(\tau^r)$ and $\alpha(\tau^c)$ respectively. We determine if a $(2i+1)\times(2i+1)$ square-prefix of any pattern occurs at τ. The text location τ generates the following tuple

$$\langle n'_e, n'_r, n'_c, T(j+2i, k+2i)\rangle$$

where n'_e, n'_r and n'_c are the prefix names of the $2i \times 2i$ square-prefix of the pattern, if any, which occur at τ, τ^r and τ^c, respectively. Note that n'_e corresponds to the prefix-name of $\alpha(\tau)$. We show how to obtain n'_c; n'_r is similarly obtained.

Consider $\alpha(\tau^c)$. If $\alpha(\tau^c)$ were a square of size $(2i-2)\times(2i-2)$ or smaller, then there does not exist any $(2i+1)\times(2i+1)$ square-prefix of P that occurs at τ. This is because, if $\alpha(\tau^c)$ were a square of size $(2i-2)\times(2i-2)$ or smaller, then the longest prefix from P can be a square of at most $(2i-1)\times(2i-1)$ in size. Note that $\alpha(\tau^c)$ can not be of size $(2i-1)\times(2i-1)$ since it is an even sided square. If $\alpha(\tau^c)$ were of size $2i\times 2i$ or larger, then determine its largest square-prefix which is a square-prefix in P^c. This is sufficient to determine n'_c, if any.

Having generated the text and pattern tuples, namestamp the set $S_t(i)$ with the set $S_p(i)$ for each i in parallel. This provides each text location τ with the prefix-name of the square-prefix of size $(2i+1)\times(2i+1)$, if any existed in P.

That completes our description of the recursive shrink-and-spawn steps for two dimensional prefix matching. For each location $T_k(i,j)$, given the pattern square-prefix of largest dimensions that matches at that location, it remains to extract the pattern array of largest dimensions that matches beginning at that location. This is performed as in the second phase described in Section 4.2. The only difference is that we look at the diagonal for determining the longest pattern that is the prefix of the given match at a text location.

We consider the complexity next. This brings up a detail concerning Steps $2a$ and $2b$. This step is not performed at each recursive level. If it were, each level of recursion would take $O(\log m)$ time, and the entire algorithm would then take $O(\log^2 m)$ time. Rather, we wait for the recursion to reach its lower most level and then in parallel this step is executed for each recursion level. Following this, "unwinding" of the recursive levels continues. Hence, over all recursive levels, these two steps take $O(\log m)$ time and $O(M)$ work. The naming in Step 1 takes $O(1)$ time and $O(n+M)$ work. As a result, the text size remains as n while the dictionary size falls to $3M/4$. The dimension of the largest pattern in P' is $m/2 \times m/2$. Step 4 takes $O(1)$ time and $O(n+M)$ work. Thus we claim,

Theorem 6 *A dictionary of two dimensional patterns of total size M such that the maximum size of any pattern is $m \times m$, can be preprocessed in $O(\log m)$ time and $O(M)$ work.*

Step 4. (Extend Right) This is the most complicated step of all.

Step 4a. Generate the prefix-names for the square-prefixes of P. Note that the square-prefixes of even dimensions are retained as square-prefixes in P''. Therefore, it remains to determine the prefix-names for the square-prefixes of odd dimensions. Consider all square-prefixes of dimension $(2i+1) \times (2i+1)$ in parallel. Consider in particular the square-prefix $P_k(1 \cdots (2i+1), 1 \cdots (2i+1))$ for some integer i.

Note that $P_k(1 \cdots 2i, 1 \cdots 2i)$, $P_k(1 \cdots 2i, 2 \cdots (2i+1))$ and $P_k(2 \cdots (2i+1), 1 \cdots 2i)$ are square-prefixes in P, P^c and P^r respectively; furthermore, they are all of even dimensions. Hence, their prefix-names have been determined recursively. Denote their prefix-names by n_e, n_c and n_r respectively. The square-prefixes of dimension $(2i+1) \times (2i+1)$ are assigned prefix-names by namestamping with the tuple

$$\langle\langle n_e, n_r, n_c, P_k(2i+1, 2i+1)\rangle, \langle k\rangle\rangle$$

generated one per each square-prefix $P_k(1 \cdots (2i+1), 1 \cdots (2i+1))$.

Step 4b. For each text location τ, the longest square-prefix from P'' that matches at τ is provided. This is a square-prefix in P'. Let this prefix be $\alpha(\tau)$. Note that this is a $2i \times 2i$ square, for some i. There are two cases.

1. If $\alpha(\tau)$ is not a square-prefix in any pattern in P, then the largest square prefix from P is the largest square prefix of $\alpha(\tau)$ that is a square prefix in P.
2. Say $\alpha(\tau)$ is a square-prefix of some pattern in P. Then, one of the following is true. Either $\alpha(\tau)$ is the largest square-prefix from P that occurs at τ or the largest square-prefix in P that occurs at τ has sides of length $(2i+1) \times (2i+1)$.

Case 1 above is taken care of as in Section 4.2. We show how to take care of the Case 2. The first task is to check if a $(2i+1) \times (2i+1)$ square-prefix of any pattern occurs at τ. If none such occurs, the longest square-prefix that matches at τ is indeed $\alpha(\tau)$. If such a square-prefix exists, then the second task is to determine the prefix-name of this square-prefix. Both these tasks are accomplished using namestamping that we describe below.

Generate a set S_p of tuples from P. The set $S_p(i)$ consists of tuples in S_p generated by the pattern positions $(2i+1, 2i+1)$ in some pattern. Consider one such position $P_k(2i+1, 2i+1)$. This location generates the following tuple

$$\langle\langle n_e, n_r, n_c, P_k(2i+1, 2i+1)\rangle, \langle k\rangle\rangle$$

where n_e, n_c and n_r are the prefix-names of the squares of size $2i \times 2i$, respectively given by, $P_k(1 \cdots 2i, 1 \cdots 2i)$, $P_k(1 \cdots 2i, 2 \cdots (2i+1))$ and $P_k(2 \cdots (2i+1), 1 \cdots 2i)$. These are square-prefixes in P' and their prefix-names have been ascertained recursively.

disjoint subarray of size 2×2 into a single symbol using names. Note that there could be a residue consisting of one row and one column of some of the pattern arrays. These residues would be considered later.

We *spawn* copies of text arrays in the following sense. Consider the set of all text arrays T' such that each text array T_i is replaced by 4 text arrays T_i^j for $1 \leq j \leq 4$. Each of these arrays is obtained from an appropriate array derived from T_i by first tiling it with subarrays of size 2×2 and subsequently replacing each subarray with its name δ'. The four arrays on which these two operations are performed so as to yield T_i^j for $1 \leq j \leq 4$ are 1. $T_i(1 \cdots t_i, 1 \cdots t_i)$, 2. $T_i(2 \cdots t_i, 1 \cdots t_i)$, 3. $T_i(1 \cdots t_i, 2 \cdots t_i)$, and 4. $T_i(2 \cdots t_i, 2 \cdots t_i)$. That is, each text array spawns off 4 copies by replacing each subarray of size 2×2 in the original text array by a single symbol.

Step 2. (Computations for Extend-Right) We perform the following two operations on P'.

Step 2a. Prefix-name the set P'.

Prefix-name for P' is computed in two steps. First, compute the prefix-names for the set of strings obtained by considering each row of each of the arrays in P' as a string. Each location (i, j) in each array P_k is associated with the prefix-name $\delta_1(P_k(i, j))$. This is the name for the prefix $P_k(i, 1) \cdots P_k(i, j)$. Consider an auxiliary array for each array $P_k \in P'$ denoted by P_k''' where $P_k'''(i, j) = \delta_1(P_k(i, j))$. Let P''' be the collection of these auxiliary arrays. Consider each column of these arrays as strings and perform another prefix-naming. Each location (i, j) in each auxiliary array P_k''' is associated with the prefix-name $\delta_2(P_k'''(i, j))$. This is the name for the prefix $P_k'''(i, 1) \cdots P_k'''(i, j)$.

Lemma 1 *The function δ_2 assigns valid prefix-names for the arrays in P'.*

Proof: Consider two rectangles, that is the $(1 \cdots i, 1 \cdots j)$ entries in two pattern arrays A and B. Their auxiliary arrays are A' and B' respectively. The claim is that $\delta_2(A'(i, j)) = \delta_2(B'(i, j))$ if and only if $A(1 \cdots i, 1 \cdots j) = B(1 \cdots i, 1 \cdots j)$. Assume $\delta_2(A'(i, j)) = \delta_2(B'(i, j))$. This implies $A'(1 \cdots i, j) = B'(1 \cdots i, j)$ since δ_2 is a prefix naming for the columns of P'. This in turn implies that $\delta_1(A(k, j)) = \delta_1(B(k, j))$ for all $k \in 1 \cdots i$. Thus $A(k, 1 \cdots j) = B(k, 1 \cdots j)$ for all $k \in 1 \cdots i$ since δ_1 is a prefix-name for the rows of P. Hence, $A(1 \cdots i, 1 \cdots j) = B(1 \cdots i, 1 \cdots j)$. The other direction is seen similarly. □

Step 2b. For each square-prefix in P', determine its longest square-prefix that is a square-prefix in P. This is done as described in Section 4.2 corresponding to Step 2 in the static dictionary matching algorithm. Similarly, determine for each square-prefix in P', its longest square-prefix that is a square-prefix in P^r. Also determine for each square-prefix in P', its longest square-prefix that is a square-prefix in P^c.

Step 3. (Recursive Step) Recursively determine, for each text location τ in T', the square-prefix of longest dimensions from P'' that matches at τ. This additionally returns the prefix-names for the square-prefixes of P''.

5 Higher Dimensional Dictionary Matching

We now discuss the two dimensional dictionary matching problem. Extensions to d-dimensional dictionary matching for a fixed d are straightforward. Consider the *static two dimensional dictionary matching problem.* We have a set $\mathcal{D}$ called the *dictionary* of pattern *arrays* $\{P_1, P_2, \ldots, P_\lambda\}$ where each P_i is a $p_i \times p_i$ square, given for preprocessing.

Input: The set T of text arrays $\{T_1, T_2, \ldots, T_\kappa\}$ where each T_i is a $t_i \times t_i$ square.

Output: For each location (i,j) in $T_k \in T$, the index of the pattern with largest sides that matches at $T_k(i,j)$.

A match between two dimensional arrays is a standard notion [B77,Ba78]. Let $\sum_{P_i \in \mathcal{D}} p_i^2 = M$ and $\sum_{T_i \in T} t_i^2 = n$. Furthermore, each $p_i \leq m$. Define a *square-prefix* of an array to be a subsquare at the top left corner. Also, prefix-names for squares are defined analogous to prefix-names for strings. We extend our algorithm for static dictionary matching with strings (from Section 4) to the case of square arrays. We first describe some details of Step 1, namely, "two dimensional prefix-matching" defined below. The input to *two dimensional prefix-matching* is the same as that for two dimensional dictionary matching. However, the output is as follows:

Output: For each text location $T_k(i,j)$ denoted by τ, compute $\delta_t(\tau)$, $|\delta_t(\tau)|$ and $\mathcal{I}(\tau)$ such that the square-prefix of largest sides from P that matches at τ is a $|\delta_t(\tau)| \times |\delta_t(\tau)|$ square-prefix with prefix-name $\delta_t(\tau)$, and $\mathcal{I}(\tau)$ is the index of a pattern with this prefix.

Suppose that each location $P_1(i,j)$ in $P_1 \in P$ is assigned its prefix-name. Formally, *prefix-naming* a set of strings S is defined as follows. Consider the set $S(i,j)$ of all the subarrays $S_k(1 \cdots i, 1 \cdots j)$ for every string $S_k \in S$. Each of the sets $S(i,j)$ is individually named. This name is called the *prefix-name.* Note that we are implicitly using a more general notion of prefix of an array being a rectangular subarray with the same left hand corner.

While solving the two dimensional prefix-matching problem, largest square-prefix that matches at a text location is specified by its prefix-name and dimensions. For the pattern arrays, it is guaranteed that the prefix-names of all the square-prefixes would have been determined. This is done recursively.

Step 1. (Two Dimensional Shrink-and-Spawn) Consider two sets P^r and P^c where P^r is obtained by stripping the top row in each pattern in P and P^c is obtained by stripping the leftmost column in each pattern in P. Let $P' = P \cup P^r \cup P^c$. Note that $|P'| \leq 3M$. Consider a naming of all subarrays of T and P' of size 2×2. Let this naming function be δ'.

We *shrink* the patterns in the following sense. Consider the set of all patterns $P'' = \{P_1', P_2', \cdots, P_\kappa'\}$ where P_k' is obtained from $P_k \in P'$ as follows. Consider each location (i,j) such that i and j are odd. Replace $P_k(i \cdots (i+1), j \cdots (j+1))$ by its name $\delta'(P_k(i \cdots (i+1), j \cdots (j+1)))$. That is, P_i' is generated from P_i by shrinking each

That completes the entire description of our algorithm. Step 1 takes $O(\log L)$ time and $O(n+ML)$ work to perform the naming. The sets T' and P' are of sizes n/L and M respectively. The length of the longest pattern in P' is m. From Theorem 1, step 2 takes $O(\log m)$ time and $O(n \log m/L + M)$ work. Step 3 takes $O(L)$ time and $O(ML + \frac{n}{L} \times L) = O(n+ML)$ work. The one time computation of all α values in Step 4 takes $O(\log m)$ time and $O(ML|\Sigma|)$ work. This is the alphabet-dependent computation in the entire algorithm. Rest of the Step 4 takes $O(L)$ time and $O(n)$ work. Hence,

Theorem 4 *A dictionary of size M and longest pattern of length at most m is processed in $O(\log m)$ time and $O(M|\Sigma|L)$ work. A text of size n can be matched against this dictionary in $O(L + \log m)$ time and $O(n \log m/L)$ work for any $L \leq \log m$.*

Corollary 1 *Let $|\Sigma| = o(\log m)$. A static dictionary can be processed in $O(\log m)$ time and $O(M\sqrt{\log m|\Sigma|})$ work. The text of size n can be matched against this dictionary in $O(\log m)$ time and $O(n\sqrt{\log m|\Sigma|})$ work.*

Corollary 2 *For any $|\Sigma$, static dictionary matching can be performed in $O(\log m)$ time. Dictionary processing involves $O(M|\Sigma|\log m)$ work and text processing involves $O(n)$ work.*

A slightly more general theorem can shown as follows. For the purposes of decreasing the alphabet-dependent complexity, consider the Extend-Left step. Encode each symbol in the dictionary using distinct binary code of length $\log|\Sigma|$. The resultant dictionary is of size $M \log|\Sigma|$. Perform operations as described earlier to move left by one bit. This takes $\log m + \log\log|\Sigma|$ time and $O(ML\log|\Sigma|)$ work. Correspondingly, to perform Extend-Left while text processing, consider the text symbols replaced by their binary codes of length $\log|\Sigma|$. As before, text processing involves moving to the left by L positions. For each of these positions, move left $\log|\Sigma|$ times, one bit at a time. It follows that,

Theorem 5 *A dictionary of size M and longest pattern of length at most m is processed in $O(\log m + \log\log|\Sigma|)$ time and $O(ML\log|\Sigma|)$ work. A text of size n can be matched against this dictionary in $O(L\log|\Sigma| + \log m)$ time and $O(n\log m/L + n\log|\Sigma|)$ work for any positive integer L.*

Assume $|\Sigma| < m$ (stronger than the Corollary 1). Set $L = \log m/\log|\Sigma|$. The time for static dictionary matching is $O(\log m)$. The work becomes $O(n\log|\Sigma| + M\log m)$. Thus text processing involves $o(n\log m)$ work while dictionary processing involves only $O(M\log m)$ work.

Step 1. (Modified Shrink-and-Spawn) Consider L copies of each pattern string in $\mathcal{D}$ obtained by successively dropping the leading symbol. Let resultant set be $\mathcal{P}$. Shrink strings in T and $\mathcal{P}$ by L to obtain text and pattern sets T' and P' respectively.

Step 2. (Dictionary Matching) Solve static dictionary matching algorithm on T' and P' using the algorithm in Section 4. Output for each text position $T_i(j)$ such that $j = kL+1$ for some integer k, is the longest prefix from P' of length a multiple of L that matches beginning at $T_i(j)$.

Step 3. (Extend-Right) For each $T_i(j)$ such that $j = kL + 1$ for some integer k, the longest prefix from P' of length a multiple of L is extended by at most $L-1$ positions to the right as in Step 3 in Section 4.1 to obtain the longest prefix from $\mathcal{P}$ that matches at $T_i(j)$. From this, determine the longest pattern that matches at $T_i(j)$ as in Section 4.2.

Step 4. (Extend-Left) Given the longest prefix from $\mathcal{P}$ that matches at $T_i(j)$ where $j = kL + 1$ for some k, extend this left and determine the longest prefix from $\mathcal{P}$ that matches beginning at $T_i(j - \mathcal{L})$ for $1 \leq \mathcal{L} \leq L - 1$. From this, determine the longest pattern that matches at $T_i(j - \mathcal{L})$.

That completes the description of our algorithm at the high level. Consider each step in some detail. The definition of set $\mathcal{P}$ is critical in being able to perform the Extend-Left efficiently. P_i^j obtained from $P_i \in \mathcal{D}$ by dropping the first j positions, is called the *j-suffix* of P_i. A *$_{\leq}k$-suffix* of P_i refers to any P_i^j for $j \leq k$. Hence, $\mathcal{P}$ is the collection of the $_{\leq}L$-suffixes of the pattern strings in $\mathcal{D}$. The first three steps follow from Section 4. We now consider the Extend-Left step. For each $T_i(j)$ such that $j = kL + 1$ for some integer k, the longest prefix of $_{\leq}L$-suffixes of $\mathcal{D}$ that match beginning at $T_i(j)$ is given as input to the Extend-Left step. Let this be $\psi(T_i(j))$. Without loss of generality, consider a fixed window $T_i(j - L) \cdots T_i(j)$ henceforth for discussions. Extend-Left is done in two steps.

Step A. For each location $T_i(j - \mathcal{L})$ compute $\alpha(\mathcal{L})$ defined iteratively as follows: $\alpha(0) = \psi(T_i(j))$ and $\alpha(\mathcal{L} + 1)$ is the longest prefix of $T(j - \mathcal{L} - 1)||\alpha(\mathcal{L})$ in $\mathcal{P}$. Here the symbol $||$ stands for string concatenation. The value $\alpha(\mathcal{L})$ satisfies the following property: the longest prefix of $\alpha(\mathcal{L})$ which is a prefix in the $_{\leq}(L - \mathcal{L})$-suffixes of $\mathcal{D}$ is the longest prefix of the $_{\leq}(L - \mathcal{L})$-suffixes of $\mathcal{D}$ that matches at $j - \mathcal{L}$.

Step B. For each location $T_i(j - \mathcal{L})$, determine the longest pattern that is the prefix of $\alpha(\mathcal{L})$. Since the longest prefix of $_{\leq}(L - \mathcal{L})$-suffixes of $\mathcal{D}$ that matches at $j - \mathcal{L}$ is a prefix of $\alpha(\mathcal{L})$, the longest pattern in particular that matches at $T_i(j - \mathcal{L})$ is a prefix of $\alpha(\mathcal{L})$.

That completes our description of the Extend-Left step. Step B follows easily from Section 4.2. It remains to demonstrate how $\alpha(\mathcal{L})$ is computed in Step A. Consider set P'' obtained from each $p \in \mathcal{P}$ by replacing it with $\sigma||p$ for each $\sigma \in \Sigma$. For each prefix in P'', compute into a table, the longest prefix from $\mathcal{P}$ as in Section 4.2. Now, string $T(j - \mathcal{L} - 1)||\alpha(\mathcal{L})$ is a prefix in P'' for any $T(j - \mathcal{L} - 1)$. Thus, $\alpha(\mathcal{L})$ is looked up from this table using namestamping.

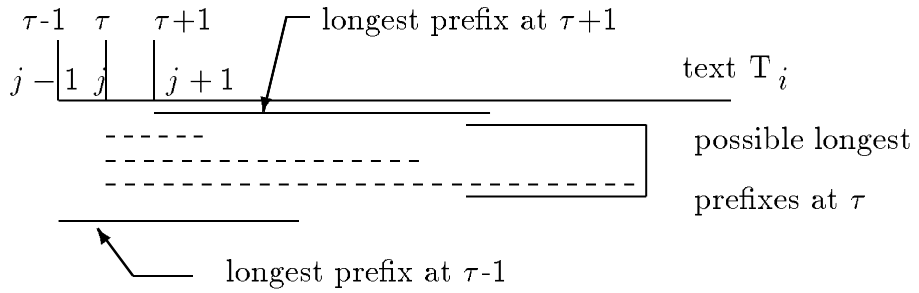

Figure 1: Longest Prefixes at Neighborhood

positions. For example, consider the scenario in Figure 1. For convenience, we refer to locations $j-1$, j, $j+1$ in text T_i by $\tau-1$, τ and $\tau+1$ respectively. Assume that the longest prefixes from the dictionary that matches at text locations $\tau-1$ and $\tau+1$ are known and from these, the longest prefix at τ need to be inferred. Clearly, the longest prefix at τ is arbitrarily long/short relative to the longest prefixes at $\tau-1$ and $\tau+1$.

As it turns out, we can relate the longest prefixes at neighboring positions, by defining the prefixes of the patterns carefully. Consider a dictionary P. Let P^s be the set of strings derived from P by replacing each string in it by its suffix obtained by deleting the leading symbol. Define $\mathcal{P}= P \bigcup P^s$. Let $\psi(\tau+1)$ denote the longest prefix from $\mathcal{P}$ that matches at $\tau+1$. Let $\phi(\tau)$ denote the longest prefix from P that matches at τ. We claim that $\phi(\tau)$ is the longest prefix of the string obtained by concatenating $\psi(\tau+1)$ to $T_i(j)$ that is also a prefix in P. Using this observation, we can efficiently compute ϕ values from the ψ values as follows. Let t be a string of the form $a||B$ (a concatenated with B) where a is a text symbol and B is a prefix in $\mathcal{P}$. Our task of computing ϕ values from ψ values is essentially that of determining the longest prefix of a string of the form t that is also a prefix in P. We accomplish the following which is clearly sufficient for this task: for every string of the form t, we determine its longest prefix that is also a prefix in P. We accomplish this by considering a set of size $|\Sigma| \times |\mathcal{P}|$ which contains all prefixes of the form t and performing a computation similar to that described in Step 2 in Section 4.2. Note that this computation is alphabet-dependent. By systematically utilizing these ideas with a variant of the shrink-and-spawn technique, we derive an algorithm for static dictionary matching which we now describe.

Our algorithm, modified from that in Section 4, is more efficient when the alphabet set Σ from which the strings are drawn is small. Our overall approach is to try and collapse the text initially to length n/L for some parameter L thereby retaining only a fraction of the text positions. Subsequently we match this shrunken text on a suitable dictionary. From the output, we construct the solution for each of the original text positions. Again for convenience, we consider that $\mathcal{D}$ and T are provided simultaneously. We proceed as described below.

4.2 Finding Longest Pattern (Step 2)

For each prefix in $\mathcal{D}$, we show how to determine its longest prefix that is a pattern. Following this, $\mathcal{I}(\tau)$ for any text location τ can be looked up from the prefix of length $|\delta_t(\tau)|$ and prefix-name $\delta_t(\tau)$ computed in Step 1, using namestamping.

1. Determine for each location $P_i(j)$ if the prefix $P_i(1)\cdots P_i(j)$ is a pattern. This is done using namestamping. The output is an auxiliary array A of size M; corresponding to each $P_i(j)$, there is a position in A that is set to 1 if $P_i(1)\cdots P_i(j)$ is a pattern and to 0 otherwise.

2. For each position $P_i(j)$, determine the largest $k \leq j$ such that $P_i(1)\cdots P_i(k)$ is a pattern. This is equivalent to finding, for each position in A, the nearest 1 to its left.

Theorem 2 *For each location $P_i(j)$ in $\mathcal{D}$, its longest prefix that is a pattern can be computed in $O(\log m)$ time and $O(M)$ operations.*

4.3 Putting It Together—Static Dictionary Matching

Recall that T and $\mathcal{D}$ were presented simultaneously. The algorithm is slightly modified if $\mathcal{D}$ is made available for preprocessing. Process the pattern strings by simulating their role in the algorithm described earlier. In a manner similar to [AILSV88,KLP89], store various tables used in the individual steps. Subsequently when T is presented, the text strings are processed by simulating the algorithm using these appropriate tables.

Theorem 3 *A set of patterns $\mathcal{D}$ each of maximum length m with total size M is processed in $O(\log m)$ time and $O(M)$ work. For each location in a text of length n, the longest pattern that matches at that location can be determined in $O(\log m)$ time and $O(n \log m)$ work.*

From our description thus far, it easily follows that while preserving other bounds, text processing can be performed with $O(n \log \Delta)$ work where $\Delta = l_{\max} - l_{\min}$ and $l_{\max}$ $(= m)$ and $l_{\min}$ denote the length of the longest and the shortest pattern respectively, in $\mathcal{D}$.

4.4 More Efficient Dictionary Matching with a Small Alphabet Size

Our algorithm in the previous section is optimal in the size of the dictionary, but is suboptimal in the text size. Intuitively, an approach towards achieving work optimality in the text size in Phase 1 of Step 1 (See Section 4.1), is to "drop" some text locations as the recursion (Step 2) progresses. The difficulty in doing this lies in inferring the longest prefix that matches beginning at each "dropped" position given those at the remaining *mental extension step*: given the prefix-name of the prefix from $\mathcal{D}$ of length $|\beta(\tau)|\ +\ \mathcal{L}$ that matches τ, determine the prefix-name of the prefix from $\mathcal{D}$ of length $|\beta(\tau)|\ +\ \mathcal{L}+1$, that matches at τ. Clearly, this incremental extension step can be used to check all possible extension lengths.

Informally, incremental extension works as follows: each prefix in the dictionary marks a table at a location indexed by its prefix-name. Each text location generates the prefix-name of the prefix of the desirable length and checks the corresponding table location to determine if there exists a prefix in $\mathcal{D}$ with that prefix-name. This procedure is implemented using the namestamping operation as described below.

Let $\mathcal{P}(T_i(j))$ denote the prefix-name of the prefix in $\mathcal{D}$ of length $|\beta(T_i(j))|\ +\ \mathcal{L}$ that matches T_i at j. The goal in incremental extension is to determine the prefix-name of the prefix of $\mathcal{D}$ of length $|\beta(T_i(j))|\ +\ \mathcal{L}\ +1$ that matches T_i at j, if any. For each location j in each text string T_i, generate a tuple

$$\langle \mathcal{P}(T_i(j)), T_i(j+|\beta(T_i(j))|+\mathcal{L}+1)\rangle.$$

Consider the set S_t of all these tuples. Partition this set into sets $S_t(\lambda)$ such that all those tuples with $|\beta(T_i(j))|+\mathcal{L}+1=\lambda$ belong to $S_t(\lambda)$. From each position r in each of the pattern strings P_j such that $r=(\mathcal{L}+1) \bmod L$, generate a tuple

$$\langle\langle\delta(P_j(r-1)), P_j(r)\rangle, \delta(P_j(r))\rangle.$$

Consider the set S_p of all such tuples. Partition this set into sets $S_p(\lambda)$ such that all pattern tuples with $r=\lambda$ belong to $S_p(\lambda)$. Namestamp set $S_t(\lambda)$ with set $S_p(\lambda)$ for each λ. We claim the following: the stamp for the tuple corresponding to a text location $T_i(j)$ is the prefix-name of a prefix from P of length $|\beta(T_i(j))|\ +\ \mathcal{L}+1$ that matches T_i at j if one such existed, and is 0 otherwise.

Implementation and Complexity.

Let $T(n,M,m)$ and $W(n,M,m)$ denote respectively, the time and work complexity of this algorithm when the text strings have total size n, the pattern strings have total size M and the length of the longest pattern string is m. The Extend-Right step takes $O(L)$ time and $O(nL+M)$ work. To sum,

$$\begin{gathered} T(n,M,m) \le \log L + T(n,M/L,m/L) + L \\ W(n,M,m) \le nL + M + W(n,M/L,m/L) + nL + M. \end{gathered}$$

Setting $L=2$ and solving,

Theorem 1 *Phase 1 of prefix-matching for a text of size n, a set of patterns each of length at most m and total size M can be solved in time $O(\log m)$ and work $O(M+n\log m)$.*

Phase 1. For each text location τ, determine $\delta_t(\tau)$, the prefix-name of the longest prefix in the dictionary that matches at τ and $|\delta_t(\tau)|$, its length.

Phase 2. Given $\delta_t(\tau)$ and $|\delta_t(\tau)|$ for each text location τ, determine $\mathcal{I}^p(\tau)$, the index of a pattern in $\mathcal{D}$ that has this prefix. This is called the *retrieve-index* problem.

We now describe Phase 1 in detail. Phase 2 is performed easily using a namestamping operation.

Algorithm Description (Phase 1)

Let L be a parameter to be fixed later. Our algorithm for Phase 1 of static prefix-matching has three steps:

1. Shrink-and-spawn Step. Shrink each string in $\mathcal{D}$ by a factor of L and spawn L copies of each string in T. The resultant set of text and pattern strings are T' and P' respectively.

2. Recursive Step. Recursively solve Phase 1 of static prefix-matching on P' and T'. For each location $T'_i(j)$ in T' denoted by τ', the output is the prefix-name of the longest prefix of any of the strings in P' that matches at τ', say $\delta_t(\tau')$, and $|\delta_t(\tau')|$, its length. Equivalently, for each text location $T_i(j)$ denoted by τ, the output is the prefix-name of the longest prefix of any of the strings in P' that matches at τ, say $\alpha(\tau)$, and $|\alpha(\tau)|$, its length.

3. Extend-Right Step. For each text location $T_i(j)$ denoted by τ, given $\alpha(\tau)$, the prefix-name of the longest prefix of any of the strings in P' that matches at τ and its length $|\alpha(\tau)|$, determine $\delta_t(\tau)$, the prefix-name of the longest prefix of any of the strings in $\mathcal{D}$ that matches at τ and its length $|\delta_t(\tau)|$.

Note that Step 1 requires naming a set of strings of length L. In Step 2, prefix-names for the strings in P' are required to solve the prefix-matching problem recursively. These can be found from the prefix-names for the strings in $\mathcal{D}$, since each string in P' is a prefix of a string in $\mathcal{D}$.

We now consider Step 3 in some detail. As before, we denote a text location $T_i(j)$ by τ for notational convenience. The longest prefix from P' that matches at τ, $\alpha(\tau)$, corresponds to a prefix in $\mathcal{D}$. Let this prefix be $\beta(\tau)$. Its length $|\beta(\tau)| = L|\alpha(\tau)|$. By the guarantee in our recursive step, no prefix in P' of length $|\alpha(\tau)| + 1$ matches at τ. This implies that no prefix in $\mathcal{D}$ of length $L|\alpha(\tau)| + L$ matches at τ. Hence, the prefix $\delta_t(\tau)$ is no more than $L - 1$ longer than $\beta(\tau)$. The task in Step 3 is to extend $\beta(\tau)$ in $\mathcal{D}$ to obtain $\delta_t(\tau)$. Let $|\delta_t(\tau)| - |\beta(\tau)|$ be the *extension length.* To determine $\delta_t(\tau)$, check for each possible extension length $\mathcal{L}$, if there exists a prefix in $\mathcal{D}$ of length $|\beta(\tau)| + \mathcal{L}$ that matches at τ as described below. Clearly, $|\delta_t(\tau)|$ is the largest $\mathcal{L}$ for which there exists a prefix in $\mathcal{D}$ of length $|\beta(\tau)| + \mathcal{L}$ that matches at τ, but no prefix of $\mathcal{D}$ of length $|\beta(\tau)| + \mathcal{L} + 1$ matches at τ. Correspondingly $\delta_t(\tau)$ is obtained.

It now remains to show for each possible extension length $\mathcal{L}$, how we check if there exists a prefix in $\mathcal{D}$ of length $|\beta(\tau)| + \mathcal{L}$ that matches at τ. Consider the following *incre-*

Fact 1 *Name-stamping set S_2 with S_1 can be done in $O(1)$ time using $|S_1| + |S_2|$ processors.*

Prefix-naming and hence naming, rely on namestamping. Prefix-naming is performed by executing a standard prefix-sum computation [FL70] using the namestamping operation in place of the standard arithmetic addition [KP88,KLP89].

Fact 2 *Given a set of strings each of length at most m and total size M, deterministic prefix-naming and hence deterministic naming, can be done in $O(\log m)$ time and $O(M)$ work.*

4 Static Dictionary Matching With Strings

Formally, the *static dictionary matching problem with strings* is as follows. A set of distinct pattern strings $\mathcal{D} = \{P_1, P_2, \ldots, P_\Lambda\}$ called the *dictionary* is available for precomputations. The *index* of the pattern P_i is i.

Input: A set $T = \{T_1, T_2, \ldots, T_\kappa\}$ of text strings.

Output: For each j, the index of the longest pattern that matches at $T_i(j)$ denoted by $\mathcal{I}(T_i(j))$.

The maximum length of any of the pattern strings is m. The size of the dictionary denoted by M, is the sum of the lengths of the individual pattern strings. The sum of the lengths of the text strings is denoted by n. We denote a text location $T_i(j)$ by τ for notational convenience. Our algorithm proceeds as described below.

Step 1: For each text location τ, determine (i.) $\delta_t(\tau)$, the prefix-name of the longest prefix in the dictionary that matches at τ, (ii.) $|\delta_t(\tau)|$, its length and (iii.) $\mathcal{I}^p(\tau)$, the index of a pattern with this prefix. This problem is called *static prefix-matching.* The shrink-and-spawn operation is applied recursively to achieve this step, as described in Section 4.1 in detail.

Step 2: Given $\delta_t(\tau)$, $|\delta_t(\tau)|$ and $\mathcal{I}^p(\tau)$ for each text location τ, determine $\mathcal{I}(\tau)$, the index of the longest pattern that matches at τ. The pattern $P_{\mathcal{I}(\tau)}$ is a prefix and the longest such, of any prefix in the dictionary of length $|\delta_t(\tau)|$ with prefix-name $\delta_t(\tau)$.

We next describe each step in detail. For convenience, assume both T and $\mathcal{D}$ are presented simultaneously. The easy modification to the case when $\mathcal{D}$ is presented before T, is explained in Section 4.3.

4.1 Static Prefix-Matching (Step 1)

Assume prefix-naming has been performed on $\mathcal{D}$. With each location j in $P_i \in \mathcal{D}$, we have its prefix-name $\delta(P_i(j))$. Prefix-matching is performed in two phases.

Output: For a naming function δ, two sets of strings U' and V' described as follows. Set V' contains the string $\delta(V_1 \ldots V_l)\delta(V_{l+1} \ldots V_{2l}) \quad \ldots \delta(V_{m-l+1} \ldots V_m)$. Set U' consists of l strings U^i for $1 \leq i \leq l$ where the string $U^i = \delta(U_i U_{i+1} \ldots U_{l+i-1})\delta(U_{l+i} \ldots U_{2l+i-1}) \ldots$. Residues of length at most $l-1$ are ignored here.

The shrink-and-spawn operation utilizes the naming function to shrink the string V by a factor of l, spawn l copies of U and maintain the following criteria. Determining all occurrences of the string V in U is equivalent to determining all occurrences of V' in U'. Hence, shrink-and-spawn essentially serves as a "match-preserving" reduction of a pattern matching problem to a "smaller" one since the length of V' is m/l. For this "match-preserving" property it is sufficient that *only* the substrings of length l which actually appear in both U and V be named using the same function δ. The substrings in U not found in V can be named using a set of special symbols distinct from the set of special symbols used to name the substrings in V which are not found in U.

3.2 Namestamping

Consider a set S_1 of distinct tuples $\langle x, y \rangle$ where y is called the *stamp* and x is called the *element*. Associate with each distinct element x in S_1, a *namestamp* denoted $l(x)$ which is the stamp of one of the tuples with element x.

Input: A set S_2 of tuples $\langle x \rangle$ where x is an element.

Output: To each tuple $\langle x \rangle \in S_2$, the namestamp $l(x)$.

Henceforth, by the phrase "namestamp set S_2 with S_1", we mean solving the above namestamping problem. Note the similarity between namestamping and table lookup. In later sections, we introduce variants of namestamping that bring out its similarity with standard dictionary operations.

3.3 Prefix-Naming

Input: A set S of strings.

Output: To each location $S_i(j)$ for some $S_i \in S$, the *prefix-name* denoted by $\delta(S_i(j))$. For each l, $\delta(S_i(l))$ is a naming function for $S_i(1)S_i(2)\cdots S_i(l)$ for each i.

Note that each distinct prefix in S is uniquely specified by its prefix-name *and* its length.

3.4 Computational Issues

In all our applications, any integer or a symbol in a string that we consider, fits into one PRAM word. Namestamping a set S_2 with S_1 in which the elements and stamps are each integers or tuples of integers $\langle a, b \rangle$ can be done using standard techniques [KP88,AILSV88,KLP89].

Our algorithms use up to m tables of size M^2 each. All of our techniques will work with space $O(M^{1+\epsilon})$ for some $\epsilon > 0$ per table, as in the work of Apostolico et al. [AILSV88]. Our work bounds include the cost for initializing these tables based on the methods of Hagerup [H88]. Randomization [GMV91] can be used to decrease the above space requirements substantially.

Parallel algorithms for dictionary matching specify the output by listing for each text location, the longest pattern from the dictionary that matches there. An alternate output format that is typically used in the sequential case is to list for each text location, all the patterns that match there; this results in an *output-bound* computation. Should this format be required in the parallel setting, even for static dictionary matching, the *interval allocation* problem [H92] seems to be inherent. Indeed, given the output of our algorithm for static dictionary matching, the algorithm of Hagerup [H93] for interval allocation can be used to output for each text location, all the patterns that match at that location; Hagerup's algorithm takes $O(\log\log^3 n)$ time and linear work.

For dynamic pattern matching, our algorithms and those due to Amir and Farach [AF91] process the initial dictionary in $O(\log|\mathcal{D}|)$ time and $O(|\mathcal{D}|\log(|\mathcal{D}|))$ work. We remark that Idury and Schaffer [IS91] have improved the sequential running time of the (initial) dictionary processing step to $O(|\mathcal{D}|)$ using quadratic space. Also, randomized algorithms for dynamic and higher dimensional dictionary matching can be found in [AFM92].

3 Basic Primitives

We introduce three important operations used in this paper.

3.1 Shrink-And-Spawn

Before defining this operation, we need to define the following additional primitive.

Naming

Input: A set S of strings of length l.

Output: For each $s_i \in S$, a $O(\log|S|)$ bit *name* denoted by $\delta(s_i)$ such that $\delta(s_1) = \delta(s_2)$ for $s_1, s_2 \in S$ if and only if $s_1 = s_2$. The function δ is a *naming function.*

The function δ is a *naming function.* For a given set, there exist several naming functions. In our applications, it is sufficient that we find *one* naming function. Using naming we now define,

Shrink-and-Spawn

Input: Two strings $U = u_1u_2\ldots u_n$ and $V = v_1v_2\ldots v_m$ and a parameter l that divides m.

and [Rab93], lead to efficient parallel algorithms only when the patterns are of equal length.

In order to cope with the more general and demanding situation wherein the dictionary consists of patterns of unequal length, we introduce the shrink-and-spawn technique that builds on the above-mentioned naming techniques. To better understand our technique, let us consider the following oversimplified yet illustrative example. We are given two strings $\mathcal{A}$ and $\mathcal{B}$ of lengths α and β respectively. The goal is to find all occurrences of string $\mathcal{A}$ in $\mathcal{B}$. Let L be the shrink (and spawn) parameter. During the shrinking step, $\mathcal{A}$ is decomposed into non-overlapping sub-strings of size L and each of these sub-strings are given names as in [KMR72].[5] The resulting shrunken string $\mathcal{A}'$ of length α/L is simply the ordered composition of names given to locations $(1, L, 2L, \ldots)$. Now in string $\mathcal{B}$, we replace each symbol with the name of the substring of L characters starting at that position, with the *same naming function* used in the context of $\mathcal{A}$. By doing this, we spawn L copies from $\mathcal{B}$ each of length β/L. Copy i is derived by composing the names given to locations $(i, i+L, i+2L, \ldots)$. By executing this step once, we have effectively reduced the size of one of the strings ($\mathcal{A}$ in this case) by a factor L without losing any of the information needed for matching. This is because in order to find matches in $\mathcal{B}$, we need to essentially consider finding matches of the (smaller) string $\mathcal{A}'$ in each of the spawned copies of $\mathcal{B}$. The overall technique involves applying this shrink-and-spawn step *repeatedly*, with appropriate choices of the parameter L. In [KLP89] this technique was implicitly used for $L = \log \alpha$ exactly *once*, to decrease the size of one of the strings initially by a $\log \alpha$ factor.

The rest of the paper is organized as follows: in Section 3, we define the basic techniques used by our algorithms. Our algorithms for static dictionary matching are described in detail, in Section 4. The main ideas in our approach to solving dictionary matching problems are highlighted in this section. In section 5, we sketch the extensions of these algorithms to higher dimensional dictionary matching. The modifications and extensions needed to cope with dynamic dictionaries are outlined in section 6. Finally, we briefly mention the optimal speed-up algorithms for problems including *multi-dimensional* pattern matching (from [KLP89]) in Section 7.

2 Model, Alphabet Size and Remarks

All our algorithms are designed using the arbitrary CRCW PRAM model[Ja92]. As in [AILSV88] and [KLP89], we are concerned with an alphabet size that is polynomial in n and M. All the bounds quoted thus far—including those for previously known algorithms [AF91]—are in the context of this alphabet size. If the input alphabet is unbounded, all known sequential and parallel algorithms for dictionary matching including that due to Aho and Corasick [AC75] perform $\Omega((n+M)\log M)$ work.

[5] We assume that α and β are multiples of L for ease of explanation; this is not a requirement in applying the technique itself.

$O(\lambda \log M)$ *amortized* work. Our parallel running time and work performed for insertions and text-matching are identical to those for partly dynamic dictionary matching stated above.

No deterministic parallel algorithms are known for this problem. The best-known sequential algorithm for implementing deletions runs in $O(\lambda \log M)$ time [AFGGP91].

4. *Prefix-matching* that we characterize in this paper plays a significant role in our approach to designing parallel algorithms for dictionary matching. Our main step is to design extremely efficient parallel algorithms for prefix-matching (Theorems 1, 7, and 9), which we then use to achieve the above-mentioned improvements for dictionary matching. This approach works since prefix-matching embodied the bottlenecks in previously known algorithms for parallel dictionary matching.

5. For the *multi-dimensional pattern matching* problem (and related problems) from [KLP89,Rab93], we present parallel algorithms with optimal speedup. In multi-dimensional pattern matching, we are given a pattern of size M and a text of size n, both of which are cubes in d-dimensions. By applying the shrink-and-spawn technique, we derive an optimal speedup parallel algorithm for this problem that runs in $O(\log m)$ time and $O(n + M)$ work. Here, $m = M^{1/d}$ denotes the number of characters in each *side* of the pattern.

   Kedem, Landau and Palem [KLP89] were the first to present an optimal speedup parallel algorithm for this, and related problems. Rabin [Rab93] presented elegant randomized algorithms for these problems—also with optimal speedup—based on fingerprinting. It is interesting to note that by using the shrink-and-spawn technique, we are able to derive deterministic algorithms that are much simpler than those from [KLP89][4] while preserving optimal speedup, *even when the overall work is linear in the input size.*

Traditionally, the approach to designing very efficient and optimal speedup parallel algorithms for string and pattern matching problems have relied on the notion of *periodicities* [Ga84,Vi85,BG90,Vi90,Ga92,AB92,ABF93,CGRMR92]. Unfortunately, these methods do not seem to scale well beyond two dimensions, or when multiple patterns are given. Therefore, Kedem, Landau and Palem [KLP89] and Rabin [Rab93] approach the problem of matching in higher dimensions very differently. Their methods and those of Apostolico et al. [AILSV88] for parallel construction of suffix trees, are inspired by the *naming* technique of Karp, Miller and Rosenberg [KMR72]. Naming involves successively refining the given set of strings into equivalence classes of increasing size. All the strings in a given equivalence class are identical, and are given a unique *name* or "certificate". Currently known parallel algorithms based on these naming constructs, from [KLP89]

[4] Their algorithm uses a sophisticated parallel construction and simulation of the Aho-Corasick [AC75] automaton which we do not need.

bet, then we improve the above algorithm to solve static dictionary matching in time $O(\log m)$ with only $O((M+n)\sqrt{\log m})$ work. (Whenever convenient, we will use the first and second terms in the expressions denoting time and work complexities to correspond to dictionary and text processing respectively.)

We note that these bounds are better than those for the best-known parallel algorithms for the above problem, even when randomization is used. Because of the manner in which the shrink-and-spawn technique works based on "naming" (to be described later), the running times and work overheads of this algorithm and its extension to dictionary matching in higher dimensions (item 2 below), are dependent only on the length of the longest pattern m rather than on M, the total size of the dictionary.

2. We extend the above algorithm to run in $O(\log m)$ time and $O(M + n \log m)$ work for dictionary matching when the patterns and text are d-dimensional, for any fixed d.

   No deterministic parallel algorithms were known previously, and the most efficient sequential algorithm from [AF92] runs in $O(M+n)$ time using quadratic space, and in $O((M+n)\log \kappa)$ time using linear space; κ denotes the number of patterns in the dictionary.

3. In the *dynamic dictionary matching problem*, we start with an initial dictionary $\mathcal{D}$ and execute a sequence of *insert*, *delete* or *match* operations that are specified *on-line*. Whereas insertions and deletions are meant to respectively add or delete a given pattern from the "current dictionary"[3] of size M, a match operation is meant to find all occurrences of patterns from it, in a given piece of text. We will use λ to denote the sizes of the patterns to be inserted or deleted, and that of the text to be matched. Amir and Farach [AF91] distinguish the *partly dynamic* version of this problem where only insert and match operations are allowed, from its *fully dynamic* variant where deletions are allowed as well.

   a. We present a parallel algorithm for partly dynamic dictionary matching that has optimal speedup for insertions and matching. Both these operations take $O(\lambda \log M)$ work and $O(\log M)$ time.
   In [AF91], Amir and Farach present a parallel algorithm that does not have optimal speedup for matching the text, since it requires $O(\lambda \log m \log M)$ work to implement this step. Also, insertions and text matching take $O(\log m \log M)$ time and hence are slower than the corresponding running times of our algorithm. The best sequential algorithm for insertions and matching is also due to Amir and Farach [AF91] and runs in $O(\lambda \log M)$ time.

   b. For fully dynamic dictionary matching, we present an optimal speedup parallel algorithm that implements the delete operation in $O(\log M)$ time and

[3] Each operation is defined on the dictionary that exists after all the other operations preceding it from the given sequence have been applied to $\mathcal{D}$, in the order specified.

1 Introduction

The input to the *dictionary pattern matching problem* is a union of distinct pattern strings represented as a dictionary $\mathcal{D}$, and a text string T. The goal is to find for each location in the text, all the patterns from the dictionary that match at that location. The classical sequential algorithm for this problem is due to Aho and Corasick [AC75] that runs in time $O(n + M)$, where n and M respectively denote the sizes of the text and the dictionary.[1] The dictionary $\mathcal{D}$ is assumed to be presented *statically* in the beginning, with the text string—or more generally a sequence of text strings—presented subsequently. Aho and Corasick preprocess the patterns in $\mathcal{D}$ and construct a tree (trie). This tree encodes a generalization of the well-known "failure" and "go-to" functions introduced by Knuth, Morris and Pratt [KMP77] in the context of *string matching*, wherein the dictionary consists of a single pattern. Unfortunately, these approaches seem to be inherently sequential and are not amenable to efficient parallelization.

The best known deterministic parallel algorithm for this problem, referred to henceforth as *static dictionary matching*, is due to Amir and Farach [AF91]. Their algorithm runs in $O(\log m \log M)$ *parallel time* and $O((M + n \log m) \log M)$ *work*, where m denotes the length of the longest pattern in $\mathcal{D}$. Using randomization, Amir, Farach and Matias [AFM92] have reported an algorithm with improved time and work bounds. Their algorithm runs in $O(\log M)$ *expected time* and performs $O((M + n) \log M)$ work. (Given an input of size N, a parallel algorithm running in time $T(N)$ using $P(N)$ processors performs work $P(N).T(N)$. A parallel algorithm has *optimal speedup* whenever it does work that is asymptotically the same as the best-known sequential algorithm that runs in $seq(N)$ steps[2] i.e., $P(N).T(N) = O(seq(N))$.)

Previously known parallel algorithms for this and other well-studied dictionary pattern matching variants [AF91,AFGGP91,AFILS93,Gi93] are not as efficient as their sequential counterparts, in that they achieve parallel speedup at the expense of performing more work. Additionally, owing to the *suffix tree* constructions used previously [AF91,GG93], the running times and the work *overhead* incurred in parallelizing depended on the size of the entire dictionary (M in the case of static dictionary matching), which can be prohibitively large.

In this paper, we present highly efficient algorithms for a range of dictionary matching problems, including the ones discussed above. We do this by introducing a general *shrink-and-spawn* technique that we apply repeatedly to design our algorithms.

1.1 Main results and Significance

1. For static dictionary matching, we present an algorithm that preprocesses the dictionary and matches the text in time $O(\log m)$. Its overall work complexity is $O(M + n \log m)$. If the text and patterns are derived from a constant-sized alpha-

[1] This bound holds for an alphabet size that is polynomial in n and M.

[2] For a sequential algorithm, its work and running time are equivalent.

Highly Efficient Dictionary Matching in Parallel*

S. Muthukrishnan† K. Palem‡

Abstract

We present highly efficient parallel algorithms for several well-studied dictionary matching problems. Our algorithms are faster and more efficient in terms of their *parallel work*, compared to previously known results.

- For *static dictionary matching*, we present an algorithm that preprocesses the dictionary and matches the text in $O(\log m)$ parallel time and $O(M + n \log m)$ work, given any dictionary of size M whose longest pattern is m characters long, and a text of size n. We have further improved this algorithm to solve static dictionary matching with only $O((M+n)\sqrt{\log m})$ work, if the characters are drawn from an alphabet of constant size. A distinguishing feature of these algorithms and the one stated below for matching in higher dimensions, is that in contrast with previous work, the running times, and work overheads when applicable, are dependent only on the length of the longest pattern m.
- We present a parallel algorithm for *d-dimensional dictionary matching* that runs in $O(\log m)$ time and matches the text in $O(M + n \log m)$ work for any fixed d.
- We present a new and more efficient parallel algorithm for *dynamic dictionary matching*. *Insertions* into and *deletions* from the dictionary, as well as *matching* the text can be done with *optimal speedup* in $O(\lambda \log M)$ work and $O(\log M)$ time. Here, λ denotes the length of the string to be inserted, deleted or matched into a dictionary of size M.

All of the above algorithms are designed by applying the *shrink-and-spawn* technique that we introduce in this paper. We also show that this technique leads to parallel algorithms that only do optimal (linear) work, for *multi-dimensional pattern matching* and related problems [KLP89,Rab93]. Our algorithms are deterministic, as those in [KLP89], but however, are much simpler and preserve the efficiency as well as the speed of those presented there.

*This research was partially supported by NSF/DARPA under grant number CCR-89-06949 and by NSF under grant number CCR-91-03953.

†Courant Institute of Mathematical Sciences, 251 Mercer Street, New York, NY 10012-1185, USA; muthu@cs.nyu.edu, (212) 998-3061.

‡IBM Research Division, T. J. Watson Research Center, P. O. Box 704, Yorktown Heights, NY 10598, USA; kpalem@watson.ibm.com, (914) 984-9846; palem@cims.nyu.edu, (212) 998-3084.