Efficient Discovery of Loop Nests in Communication Traces of Parallel Programs

Qiang Xu, Jaspal Subhlok

Computer Science Department
University of Houston
Houston, TX, 77204, USA
http://www.cs.uh.edu

Technical Report Number UH-CS-08-08
May 26, 2008

Keywords: Trace Compression, Crochemore's algorithm, Performance analysis and modeling

Abstract

Execution and communication traces are central to performance modeling and analysis of parallel applications. Since the traces can be very long, meaningful compression and extraction of representative behavior is important. Commonly used compression procedures identify repeating patterns in sections of the input string and replace each instance with a representative symbol. This can prevent the identification of long repeating sequences corresponding to outer loops in a trace. This paper introduces and analyzes a framework for identifying the maximal loop nest from a trace based on Crochemore's algorithm. The paper also introduces a greedy algorithm for fast ``near optimal" loop nest discovery with well defined bounds. Results of compressing MPI communication traces of NAS parallel benchmarks show that both algorithms identified the basic loop structures correctly. The greedy algorithm was also very efficient with an average processing time of 16.5 seconds for an average trace length of 71695 MPI events.

Efficient Discovery of Loop Nests in Communication Traces of Parallel Programs

Qiang Xu
University of Houston
Qiang.Xu@mail.uh.edu

Jaspal Subhlok
University of Houston
jaspal@uh.edu

Abstract

Execution and communication traces are central to performance modeling and analysis of parallel applications. Since the traces can be very long, meaningful compression and extraction of representative behavior is important. Commonly used compression procedures identify repeating patterns in sections of the input string and replace each instance with a representative symbol. This can prevent the identification of long repeating sequences corresponding to outer loops in a trace.This paper introduces and analyzes a framework for identifying the maximal loop nest from a trace based on Crochemore's algorithm. The paper also introduces a greedy algorithm for fast "near optimal" loop nest discovery with well defined bounds. Results of compressing MPI communication traces of NAS parallel benchmarks show that both algorithms identified the basic loop structures correctly. The greedy algorithm was also very efficient with an average processing time of 16.5 seconds for an average trace length of 71695 MPI events.

Index Terms

Trace compression, Crochemore's algorithm, Performance analysis & modeling

1 Introduction

Execution and communication traces are central to performance analysis and performance modeling of parallel applications. However, trace processing is a challenge as the trace length can be large even for traces of relatively coarse grain events. Fortunately execution traces often contain repeating sequences that can be identified to capture representative behavior. The specific context of this research is the construction of performance skeletons of parallel applications for performance prediction [14, 16, 20]. A performance skeleton is a short running program that recreates the computation and communication behavior of the original application execution. A key step in the process of construction of performance skeletons is the identification of repeating patterns in MPI message communication. Since the MPI communication trace is typically a result of loop execution, discovering the executing loop nest from the trace is central to the task of skeleton construction.

The goal of the research presented in this paper is to develop effective and efficient procedures to identify the representative sections of an execution trace by discovering the loop nest structure inherent in the trace. There are, of course, several well known algorithms and tools for string compression based on substring matching. The major challenge in achieving maximal compression is discovery of long range repeating patterns, typically representing an outer loop in an execution trace. It is normal that many overlapping repeating substrings of different lengths exist in an execution trace. Most compression procedures apply heuristics to selectively reduce sets of repeating substrings. Examples include *gzip* that constructs a dictionary of frequently occurring substrings and replaces each occurrence with a representative symbol, and *Sequitur* that infers the hierarchical structure in a string by automatically constructing and applying grammar rules for reduction of substrings. While these approaches can be efficient and procedures can be designed to have execution time that is nearly linear in trace length, they are not guaranteed to identify long range loop patterns because of early reductions. An alternate approach is to attempt to identify the longest matching substring first. However, simple algorithms to achieve this are quadratic for a given match length and hence cubic in trace length, and therefore impractical for long traces. A practical approach is to limit the window size for substring matching, which again risks missing long span repeats.

Before presenting our approach, we introduce the basic terminology for distinguishing various types of repeating patterns. Repeating substrings (or *repeats*) in a string can be *tandem* repeats where successive repeat substrings immediately follow each other, *overlapping* repeats where repeat substrings overlap, and *split* repeats where repeat substrings are separated by other symbols. Since we seek to identify the loop structure in a trace, we are only interested in tandem repeats. A tandem repeat is *primitive* if it is itself not composed of tandem repeats of another substring. A set of tandem repeats is *maximal* if there is no identical substring immediately preceding or succeeding the sequence of tandem repeats. We will refer to the primitive and maximal tandem repeats in a string as *PM-repeats*. In the rest of the paper, discovery of “loops” technically refers to the discovery of PM-repeats in the execution trace, which (presumably) exist because of execution of program loops. Our objective is finding and reducing the PM-repeats of different spans in an execution trace, which is the same as discovering the inherent loop nest structure in the execution trace.

To illustrate the properties of PM repeats, consider the string *abababab*. The PM-repeats corresponding to this string are represented as $(ab)^4$ which is the most compact representation. The string can also be represented as tandem repeats $(abab)^2$ but this would not be primitive, since the repeating substring itself is a tandem repeat of another string ab. The string can also be represented as $(ab)^3ab$ but this would not be a maximal repeat. Hence, $(ab)^4$ represents the only PM-repeats sequence, or optimal loop, for this string.

Our approach to identifying the loop structure in a trace is based on Crochemore’s algorithm [3], which can identify all repeats in a string, including tandem, split, and overlapping repeats, in $O(nlogn)$ time. A framework was developed in this research to discover the loop nest structure by recursively identifying the longest span tandem repeat in a trace. The procedure identifies the optimal (or most compact) loop nest in terms of the span of the trace covered by loop nests and the size of the compressed loop nest representation. The procedure was applied to identify the loop nests in the MPI communication traces of NAS benchmarks. The compression results were very good, but the execution time was unacceptable for long traces; processing of a trace consisting of approximately 320K MPI calls took over 31 hours.

The results motivated us to develop a greedy procedure for the discovery of the loop structure, which is the most important contribution of this paper. The greedy procedure intuitively works bottom up - it identifies and reduces the shorter span inner loops and replaces them with a single symbol, before discovering the longer span outer loops. In this respect, it appears similar to other approaches that apply heuristics to identify repeating substrings and replace them with symbols to enable efficient processing. However, the key characteristic of our algorithm is that only primitive and maximal tandem repeats (PM-repeats) representing a section of the trace that corresponds to loop execution, are reduced to a single symbol. No other repeating substrings are reduced. The intuition is that reduction of trace sections corresponding to complete inner loop execution will not interfere with the discovery of outer loops.

The key analytical result in this work is that the reduction of a shorter span PM-repeats (inner loops) can impact the discovery of a longer span PM-repeats (outer loops) only in the following way: if the optimal outer loop is L_o then a corresponding loop L_g will be identified despite the reduction of an inner loop. L_o and L_g have identical number of loop elements, but L_g may have up to 2 less loop iterations than L_o. Hence, the loop structure discovered by the greedy algorithm is *near optimal*.

The greedy loop nest discovery procedure was also implemented and employed to discover the loop nests in the MPI traces of NAS benchmarks. The loop nests always satisfied the criteria above, and were, in fact, identical to the optimal loop nests in all but one case. However, the time for loop discovery was dramatically lower than the optimal algorithm, with the compression time reduced to approximately 62 seconds from 31 hours for one trace.

To the best of our knowledge, this is the first effort toward extracting complete loop nests from execution traces. The paper presents detailed results on the effectiveness of these algorithms in discovering loop nests and achieving compression. The performance and scalability of the greedy and optimal algorithms are also presented and analyzed. Of particular interest are the insights into the theoretical complexity of the algorithms and the empirical measurements of performance. The methodology developed is applicable to any sequence that is likely to contain a loop structure even though the experimental results presented in this paper are limited to message passing communication traces.

The rest of the paper is organized as follows. After a discussion of related work in Section 2, we present and analyze the optimal and greedy compression procedures in Section 3 and Section 4, respectively. Experimental results and discussion are presented in Section 5. Finally, we conclude in Section 6.

2 Related work

Compression is a basic operation in a wide variety of scenarios. Many algorithms have been developed for text compression and employed in utilities like *gzip* [21]. The basic approach in such algorithms is to identify recurring short strings and replace them with identifiers. *Sequitur* [12, 11, 10] is a well-known algorithm that was developed to discover the natural hierarchy in text and other data. The insight is that repeating substrings are replaced by a grammar rule that generates that substring and the process is continued recursively, resulting in a hierarchical representation of the structure of the string. In order to improve the processing time and quality of compression, PGTC (path grammar guided trace compression) [4] is proposed as an enhanced approach that employs program static analysis to build a grammar and guide compression.

The goal of our compression approach is to identify complete loop nests from the repeating substrings discovered in a string. Replacement of short substrings with identifiers, as is the case in above methods, can result in failure to identify long span repeats in traces corresponding to outer loops.

Noeth et al. [13] have employed a scalable trace driven approach to analyze MPI communication, which is based on identifying loops from a message passing trace. The compression algorithm maintains a queue of MPI events and attempts to greedily compress the first matching sequence within a sliding window. They extend regular section descriptors (RSDs) for single loops to express MPI events nested in a loop in constant size [7] while power-RSDs are utilized to recursively specify RSDs nested in multiple loops [8]. The drawback is that the algorithm is not guaranteed to yield the optimal loop nest as matching is limited to a maximum sliding window to avoid $O(n^2)$ time complexity in the length of the trace.

There are two well known approaches to identifying all repeats in a string systematically - one based on suffix trees and the other based on Crochemore's algorithm. A compression scheme based on Crochemore's algorithm that uses split and tandem repeats to carry out offline genetic data compression is proposed in [17]. We have employed Crochemore's algorithm as the basis of our approach and that is discussed in detail in this paper. We briefly discuss suffix trees here.

Suffix trees are a fundamental data structure supporting a wide variety of efficient string searching algorithms. In particular, suffix trees are well known to allow efficient and simple solutions to problems concerning the identification and location of repeated substrings. Several algorithms [19, 9, 18] can build a suffix tree in linear time. Stoye and Gusfield have developed an $O(n \log n)$ time method [15] to find all occurrences of primitive tandem repeats in a string with suffix trees. They also proposed a novel method [6] to collect only the primitive tandem repeat $types$ in $O(n)$ time and find occurrences of all primitive tandem repeats in $O(n + z)$ time, where z is the number of occurrences of primitive tandem repeats in a string. In [1], the repeating substrings in a string and their statistics are inferred from suffix trees, and used for compression through greedy off-line textual substitution.

We have based our loop nest identification procedure on Crochemore's algorithm instead of suffix trees for two main reasons. First, we are not aware of a straightforward approach to finding all $primitive$ and $maximal$ tandem repeats with suffix trees. Second, the process of building and processing suffix trees is significantly more complex than that based on Crochemore's algorithm.

Another dimension of compression of traces from parallel programs is inter-node compression. The traces from different processes in the system can be consolidated into a single trace before compression [20, 2] or after compression [13]. This aspect is orthogonal to the work proposed in the paper as the trace compression procedures can be applied to a consolidated logical trace or a single process trace.

3 Optimal trace compression

The main contribution of this paper is a framework to compress execution traces by discovering the loop structure inherent in the trace. All repeating substrings in a trace are identified by employing the well known Crochemore's algorithm. However, the repeats are implicit in a complex data structure. The total number of repeats can be combinatorial in the size of a string and very large in practice. **Our contribution is the development of a framework to efficiently construct the loop structure in the trace by selectively filtering and reducing the repeats.** The procedure consists of the following steps for discovery and reduction of outermost loops:

1. **Repeats discovery:** Discovery of all types of repeats (overlapping, split, and tandem) of all sizes by Crochemore's algorithm.

2. **Loop identification:** Identification of all PM-repeats (primal and maximal tandem repeats) corresponding to loops.

3. **Loop filtering:** Discovery of outermost loops and their replacement with loop symbols.

The above process is repeated recursively inside each discovered loop. For a string with n symbols, the repeats discovery takes $O(n \log n)$ time while loop identification and loop filtering take $O(n^2)$ time. Hence the overall complexity is $O(n^2)$. We discuss each of the above steps and the overall loop identification and compression procedure.

3.1 Repeats discovery

As an optimized **successive refinement method**, Crochemore's algorithm [3] computes all repeating substrings (tandem, overlapping, and split) in a finite string S of length n in $O\,(n \log n)$ time. The successive refinement begins with grouping all positions in the string that have the same symbol/character into a single class. Each class is then refined into new subclasses that contain starting positions of repeating substrings of length two. The process is continued to find the starting position of all repeating substrings of length, 3,4,5....until a size is reached for which no repeating substrings exist.

Before describing the details of Crochemore's algorithm, we introduce some basic string definitions.

Definition: A *string* $S = s_1 s_2 s_3 ... s_n$ is an ordered list of characters/symbols written contiguously from left to right. The *length* of *S* is $|S|$. $S[i..j]$ is the *substring* of *S* that starts at position *i* and ends at position *j*.

Definition: E_k is an *equivalence relation* over a string S defined as follows: iE_kj if and only if substrings $S[i..i+k]$ and $S[j..j+k]$ are identical. E_k partitions the positions of string S into equivalence classes; if iE_kj then i and j will be in the same E_k class. We also use E_k to denote the set of those equivalence classes.

The simple successive refinement is based on the fact that for string S, if iE_kj and $S[i+k]$=$S[j+k]$, then $iE_{k+1}j$. For example, consider the string

S =	a	b	a	a	b	a	b	a	a	b	a	a	b	$
	1	2	3	4	5	6	7	8	9	10	11	12	13	14

Initially, we construct three E_1 classes containing repeating substrings of length one. Note that the unique character "$" is appended as the end of string symbol.

E_1 :	{1,3,4,6,8,9,11,12}	{2,5,7,10,13}	{14}
	a	b	$

The first step of refinement splits each class of E_1 into classes that contain starting positions of substrings of length two. We check the character following each position in that class. For class a–{1,3,4,6,8,9,11,12}, we need to check positions {2,4,5,7,9,10,12,13} in S. Since $S[4]$=$S[9]$=$S[12]$=a

and $S[2]$=$S[5]$=$S[7]$=$S[10]$=$S[13]$=b, class a is split into subclass aa–{3,8,11} and subclass ab–{1,4,6,9,12}. Similarly, class b–{2,5,7,10,13} is split into subclass ba–{ 2,5,7,10} and subclass $b\$$–{13}. The class $–{14} has no substring of length two starting from it, so it is discarded.

The successive refinement continues with checking of the kth character following the positions in each class of E_k to construct E_{k+1}. Any singleton classes are discarded. Eventually a value of k is reached for which there are no classes of E_k and the process is terminated. The entire process of refinement of string $S = abaababaabaab\$$ is shown in Table 1.

Table 1: Successive Refinement of String $abaababaabaab\$$

$Level1$ –	E_1 :	{1,3,4,6,8,9,11,12} a	{2,5,7,10,13} b	~~{14}~~ $\$$		
$Level2$ –	E_2 :	{1,4,6,9,12} ab	{3,8,11} aa	{2,5,7,10} ba	~~{13}~~ $b\$$	
$Level3$ –	E_3 :	{1,4,6,9} aba	~~{12}~~ $ab\$$	{3,8,11} aab	{2,7,10} baa	~~{5}~~ bab
$Level4$ –	E_4 :	{1,6,9} $abaa$	~~{4}~~ $abab$	{3,8} $aaba$	~~{11}~~ $aab\$$	{2,7,10} $baab$
$Level5$ –	E_5 :	{1,6,9} $abaab$	~~{3}~~ $aabab$	~~{8}~~ $aabaa$	{2,7} $baaba$	~~{10}~~ $baab\$$
$level6$ –	E_6 :	{1,6} $abaaba$	~~{9}~~ $abaab\$$	~~{2}~~ $baabab$	~~{7}~~ $baabaa$	
$Level7$ –	E_7 :	~~{1}~~ $abaabab$	~~{6}~~ $abaabaa\$$			

The total running time for the algorithm as described above is $O(n^2)$, since there can be $O(n)$ levels and each level can take $O(n)$ time. But two techniques proposed in [3] optimize the successive refinement and reduce the running time to $O(n \log n)$. We mention them very briefly here.

The first technique is "***indirect refinement***". This is based on the observation that any class of E_{k+1} is a subset of some E_k class. Because, $iE_{k+1}j$, if and only if iE_kj and $i + 1E_kj + 1$, we can use classes at the same level to carry out successive refinement instead of referring back to the original string S. This helps in reducing the complexity when used with another technique called " ***small classes***", which can be outlined as follows. The indirect refinement process for an E_k class into E_{k+1} classes requires matching against several, but not all, other E_k classes. The small classes techniques prescribes that the classes be selected in a specific manner that favors matching against smaller classes, and thereby avoiding some matching against larger classes. This description only gives a flavor of these technique and the interested reader is referred to [5, 3] for details. Our implementation includes the indirect refinement and small classes techniques.

3.2 Loop identification

The repeats discovered by Crochemore's algorithm include tandem, overlapping, and split repeats. Their definitions are as follows:

Definition For positions i and j of string S, that belong to the same E_k class, if $|j-i|=k$, then repeating substrings $S[i..i+k-1]$ and $S[j..j+k-1]$ are $tandem$ repeats; if $|j-i|<k$ they are $overlapping$ repeats; and if $|j-i|>k$ they are $split$ repeats.

For example, the substring aba repeats four times in string S at positions 1,4,6, and 9 in Table 1. The second aba is right behind the first one, so they constitute **tandem** repeats. The second and third aba are **overlapping** repeats, while the first and third aba are **split** repeats.

Our goal here is to find loop structures, so we need to identify and report only the tandem repeats. Tandem repeats in a string can be represented by a triple (i, β, l), where i is the starting position, β is the repeated substring, and l is the number of iterations. But a substring may be represented by multiple tandem repeats. For example, the string $abababababababab$, could be described as $(1, ab, 8)$, or $(1, abab, 4)$, or $(1, abababab, 2)$. Clearly the loop that we would like to identify corresponds to $(1, ab, 8)$. To generalize, we define **PM-repeats** and a corresponding **PM-triple**, where P and M stand for $primitive$ and $maximal$. A triple (i, β, l) corresponds to a primitive tandem repeats sequence if and only if β is not periodic. The triple corresponds to a maximal tandem repeats sequence if and only if there is no β right before or after the repeats. So, the above string can be represented by a unique PM-triple, $(1, ab, 8)$. (A PM-triple is a representation of a PM-repeats sequence and we will use the terms interchangeably.)

For each E_k class refined in Crochemore's algorithm, a PM-triple can be identified by the following Lemma, which is also mentioned in [5]:

Lemma 1. *(i,β, l) is a PM-triple, where β is a k-length substring, if and only if some single class of E_k contains a maximal series of numbers i, i+k, i+2k, ..., i+lk, such that each consecutive pair of numbers differs by k.*

In order to identify loops, PM-triples must be identified at each level during the execution of Crochemore's algorithm. Since the total number of members in all classes at a level k is bounded by string length n, the process takes $O(n)$ time. Since the maximal possible size of β, the loop element, is half the length of the string n, we need to report PM-triples after discovering the repeats by Crochemore's algorithm till level $n/2$. The running time for identifying loops represented by PM-triples is $O(n^2)$.

3.3 Loop filtering

The previous steps provide a list of PM-triples which represent all the loops in the trace. Our interest is in finding all the outermost or longest span loops. These are represented by the PM-triples at the highest level. (The inner loops are discovered by running the entire process recursively). In case of multiple overlapping loops of equal span at the same level, we select the one that starts earliest in the string.

As an example, for the string $abcdabcdabcdabcda$, 4 PM-triples will be identified. These PM-triples are $(1, abcd, 4)$, $(2, bcda, 4)$, $(3, cdab, 3)$ and $(4, dabc, 3)$. The first two PM-triples both have a span of 4 versus a span of 3 for the remaining two. Based on the earliest starting point, the selected loop will be $(1, abcd, 4)$.

As another example, consider the string $EabababababFEababababFEababababF$, which contains a loop nest containing loops at two levels. The PM-triple $(1, EabababababF, 3)$ represents the selected outer loop. The inner loops represented by triples $(2, ab, 4)$, $(12, ab, 4)$, and $(22, ab, 4)$ are ignored at this stage.

The loop filtering step repeatedly finds the PM-triple corresponding to the longest span loop, until no PM-repeats are left. Since the loops can theoretically be as small as 2 elements, the theoretical upper bound of this step with a simple implementation is $O(n^2)$. An $O(nlogn)$implementation is possible. However, in practice this is a very quick step as the number of loops is normally very small, and $O(n^2)$ is a very loose upper bound.

3.4 Compression framework summary

The procedure discussed in this section identifies all outermost loops represented in a trace. The algorithm runs recursively on the substrings that constitute the loop elements (or body) for identification of the inner loops. The recursive steps are important to get a high degree of compression. It is theoretically possible to reuse some of the information from discovery of outer loops to identify inner loops. In practice, this is likely to make little difference in performance since the processing time is dominated by the time to discover outer loops. In order to develop a compressed representation, the loop spans in the trace are replaced with tuples (LE, l), where LE is the loop element symbol, and l is the number of loop iterations. A separate table is constructed, which maps a loop element symbol to the substring that constitutes the loop element. The overall complexity of the steps in this procedure is $O(n^2)$ and is dominated by the loop identification step.

4 Greedy trace compression

The scheme discussed in Section 3 discovers the optimal loop nest. However, the execution time for loop discovery can be high for long traces. We will discuss experimental results in detail later in this paper, but we take a look now at Table 4 to motivate the case for a faster approach. As an example consider the class C SP benchmark in the table. The total time to run the algorithm is around 747 seconds, although all program loops had already been discovered in just 5.8 seconds. The reason is that the largest loop consists of only 67 symbols while the trace size is 26888 symbols. Our approach builds equivalence classes and looks for loops in increasing order from 1 to half the trace length. Even though all loops were discovered by the time the equivalence class of 67 was constructed, (and these loops spanned over 99% of the trace) there is no way for the algorithm to be certain that a larger loop does not exist, and hence refinement continues until the equivalence class of 13444 that corresponds to half the original string size. If the loops already discovered were replaced by a single symbol at the equivalence class of 67, the trace size to be processed would be less than 1% of the original trace size, and the remaining processing would be much faster. This motivates greedy compression.

4.1 Greedy trace compression

In the compression framework of Section 3, actual compression happens only after the discovery of all PM-triples, which involves the successive refinement and identification of the tandem repeats until the level of half the original trace size. The key idea of greedy compression is early compression as PM-triples are discovered. The entire span of the corresponding loop is replaced by a single symbol, and compression continues on the newly formed (shorter) string. The procedure continues until half the current string size is reached. Note that in the greedy approach, the string size decreases dynamically as loops are discovered, which is the key reason for improved performance. Figure 1 outlines the greedy compression procedure.

```
S = string corresponding to the original trace
$Current\_S = S$
$Level = 1$

Step 1:
if $Level > |Current\_S|/2$ then
   Goto Step 3
else
   i) Find all repeats of size $Level$ in $Current\_S$ by successive refinement with Crochemore's
   algorithm.
   ii) Identify all PM-triples with repeating substring of size Level.
   if any PM-triples with repeating substring of size $Level$ are discovered then
      Goto Step 2
   else
      $Level = Level + 1$; Goto Step 1
   end if
end if

Step 2:
Update $Current\_S$ by reducing all PM-triples with repeating substrings of size $Level$ in de-
creasing order of loop span, employing filtering (discussed in section 3.3) for overlapping triples.
The symbols replacing the PM-triples are stored in a mapping table.
$Level = 1$; Goto Step 1.

Step 3:
Stop. $Current\_S$ along with the mapping table for symbols is the compressed trace that captures
the loop nests.
```

Figure 1: Greedy compression procedure

The worst case time complexity for this greedy algorithm is the same as the optimal algorithm discussed in Section 3. In fact, the two algorithms will run identically if there were no PM-repeats in a trace. However, the greedy algorithm is much more efficient in practice for programs with a loop structure as it does not need to perform repeats discovery and loop identification across long spans of a trace. In our experience, almost all traces generated from executing programs have the bulk of the trace included in loop nests.

4.2 Risk of greedy compression

For most traces that we have analyzed, the greedy and optimal procedures yield identical results. Here we illustrate with carefully selected examples how the results of greedy compression can be suboptimal.

Consider the string $abaababaabaab$. The greedy compression proceeds as follows:

a b a a b a b a a b a a b $

a b L_1 b a b L_1 b L_1 b $ $L_1 = (a)^2$

a b L_1 b a L_2 b $ $L_2 = (bL_1)^2 = (baa)^2$

The loop structured discovered by the greedy procedure is $ab(a)^2ba(b(a)^2)^2b$ whereas the optimal loop structure is $(ab(a)^2b)^2(a)^2b$. A 2 iteration loop with the largest element, $(abaab)^2$, is completely missed.

Now consider the string $abaababaababaabaab$. The greedy compression proceeds as follows:

a b a a b a b a a b a b a a b a a b $

a b L_1 b a b L_1 b a b L_1 b L_1 b $ $L_1 = (a)^2$

a b L_1 b a b L_1 b a L_2 b $ $L_2 = (bL_1)^2 = (baa)^2$

L_3 a L_2 b $ $L_3 = (abL_1b)^2 = (abaab)^2$

The loop structured discovered by the greedy procedure is $(ab(a)^2b)^2a(b(a)^2)^2b$ whereas the optimal loop structure is $(ab(a)^2b)^3(a)^2b$. The loop with the largest loop element $(abaab)$ is captured, but with one less iteration than optimal.

4.3 Bounds on greedy compression results

As is the case with other heuristic algorithms discussed in Section 2, this greedy algorithm may not discover the optimal loop nest. However, the algorithm is selectively discovering and replacing loops corresponding to PM-repeats, and not any other repeat pattern. The insight is that loops are either nested or disjoint, hence a reduction of all sequences corresponding to a complete inner loop will not prevent a longer span outer loop from being identified. However, this is not completely true. Loops as recognized by PM-repeats can overlap, but only at the boundary iterations. The general informal result is as follows:

The early reduction of all inner loops corresponding to a fixed loop body can impact the identification of a longer span outer loop only as follows: the body of the discovered loop may be a reordering of the body of the original loop, and the number of iterations in the discovered loop may be up to 2 fewer than the number of iterations in the original outer span loop.

The result is formally discussed in the appendix of this paper. In other words, the discovered outer loop may start at a different point in the trace and up to 2 iterations may be lost due to reduction of an inner loop. However, the basic loop structure will be identified, unless the outer loop consists of only 2 or 3 iterations. For most communication traces, each loop typically iterates a large number of times, hence missing one or two iterations or completely missing a 2 or 3 iteration loop is not a significant practical problem.

5 Experiments and results

A framework for loop discovery and compression discussed in Section 3 and a framework for greedy compression as discussed in Section 4 were implemented. The goal was to validate loop discovery and compression achieved by these algorithms and study the tradeoffs between the execution performance and the degree of compression.

5.1 MPI communication traces

This research was motivated in the context of analysis of execution traces of MPI applications to build representative performance skeleton programs [14]. The process consists of collection of traces, *logicalization* whereby a single logical trace represents SPMD execution, compression and identification of representative sections, and generation of an executable skeleton program [20].

All results presented in this paper are for compression of MPI communication traces for Class B/C NAS Parallel Benchmarks running on 16 nodes. The traces were collected with the PMPI library and the process of conversion of a trace to a string of symbols is illustrated in Table 2.

Table 2: Conversion of MPI communication trace to a string of symbols

Raw Trace	**MPI Event**	**Symbol**
#Generating Logfile		
Node=0 #939220507ss#0#939220507		
1. 2#1#3220724724#1#28#0#134#0#939220509#939220509	[*MPI_Bcast(...1, MPI_INT, 0,...)*]	**M4**
2. 2#2#136373224#1#27#0#134#0#939220509#939220509	[*MPI_Bcast(...1, MPI_DOUBLE, 0,...)*]	**M5**
3. 2#3#135838396#3#28#0#134#0#939220509#939220509	[*MPI_Bcast(...3, MPI_INT, 0,...)*]	**M6**
4. 7#1#135789088#360#27#1#3000#138#153016848#0#939220509#939220509	[*MPI_Irecv(... 1, MPI_DOUBLE, 360, ...)*]	**P1**
5. 7#2#135786208#360#27#1#2000#138#153017012#0#939220509#939220509	[*MPI_Irecv(... 1, MPI_DOUBLE, 360, ...)*]	**P1**
6. 7#3#135794848#360#27#2#5000#138#153017176#0#939220509#939220509	[*MPI_Irecv(... 2, MPI_DOUBLE, 360, ...)*]	**P7**
7. 7#4#135791968#360#27#2#4000#138#153017340#0#939220509#939220509	[*MPI_Irecv(... 2, MPI_DOUBLE, 360, ...)*]	**P7**
8. 7#5#135800608#360#27#3#6000#138#153017504#0#939220509#939220509	[*MPI_Irecv(... 3, MPI_DOUBLE, 360, ...)*]	**P4**
9. 7#6#135797728#360#27#3#7000#138#153017668#0#939220509#939220509	[*MPI_Irecv(... 3, MPI_DOUBLE, 360, ...)*]	**P4**
10. 9#1#135812616#360#27#1#2000#138#153002824#0#939220509#939220509	[*MPI_Isend(... 1, MPI_DOUBLE, 360, ...)*]	**P10**
11. 9#2#135809736#360#27#1#3000#138#153002964#0#939220509#939220509	[*MPI_Isend(... 1, MPI_DOUBLE, 360, ...)*]	**P10**
12. 9#3#135818376#360#27#2#4000#138#153003104#0#939220509#939220509	[*MPI_Isend(... 2, MPI_DOUBLE, 360, ...)*]	**P16**
13. 9#4#135815496#360#27#2#5000#138#153003244#0#939220509#939220509	[*MPI_Isend(... 2, MPI_DOUBLE, 360, ...)*]	**P16**
14. 9#5#135824136#360#27#3#7000#138#153003384#0#939220509#939220509	[*MPI_Isend(... 3, MPI_DOUBLE, 360, ...)*]	**P13**
15. 9#6#135821256#360#27#3#6000#138#153003524#0#939220509#939220509	[*MPI_Isend(... 3, MPI_DOUBLE, 360, ...)*]	**P13**
16. 22#1#12#153016848#153017012#153017176#153017340#153017504 #153017668#153002824#153002964#153003104#153003244#153003384 #153003524#0#939220509#939220513	[*MPI_Waitall(...)*]	**O2**
17. 9#7#135786208#1470#27#1#3000#136#153003524#0#939220513#939220513	[*MPI_Isend(... 1, MPI_DOUBLE, 1470, ...)*]	**P11**
18. 7#7#135809736#1470#27#1#3003#136#153017668#0#939220513#939220513	[*MPI_Irecv(... 1, MPI_DOUBLE, 1470,...)*]	**P2**
19. 21#1#153003524#0##939220513#939220513	[*MPI_Wait(...)*]	**O1**
20. 21#2#153017668#0##939220513#939220513	[*MPI_Wait(...)*]	**O1**
....	....	**...**
2277. 3#1#3220724688#3220724696#1#27#100#0#134#0#939220642#939220642	[*MPI_Reduce(...1, MPI_DOUBLE, MPI_MAX, ...)*]	**M3**
2278. 1#2#91#0#939220642#939220642	[*MPI_Barrier*]	**M2**
#Finished writing logfile for node=0#939220642#939220646		
Trace as a string of symbols is as follows: {**M4,M5,M6,P1,P1,P7,P7,P4,P4,P10,P10,P16,P16,P13,P13,O2,P11,P2,O1,O1, ... ,M3,M2**}		

Each symbol in the trace corresponds to an MPI operation and corresponding parameters, e.g, an "*isend* operation for *4KBytes* of data to *node 2*". A simple linear search was used to find the symbol corresponding to a trace entry in the symbol table, although a hash table would be more appropriate if the number of distinct MPI operations was large. Also, similar MPI operations (e.g. 2 message sends to the same node but with slightly different size) may be combined to provide a higher degree of compression with some lossyness [14], but this was not done for the experiments in this paper. All results presented are for the compression of the string of symbols derived from the traces of NAS benchmarks.

5.2 Results and discussion

Table 3 shows the results of the optimal compression procedure. We observe that the length of the traces ranged from 8909 to 323048 (average 71695) and the length of the compressed traces ranged from 10 to 648 (average 165) with the degree of compression ranging from 15 to 5127 (average 1815). The structures of the major loop nests discovered are also described in the table. Most of the trace was covered by loops for all benchmarks, 98% on average. The conclusion is that MPI traces typically have a loop structure that can be discovered automatically by this approach. The degree of compression is excellent and the length of the compressed trace is typically relatively small.

Table 3: Results of optimal compression

Name	Trace Length	Major Loop Structure	Trace Span Covered by Loops	Compressed Trace Length	Compression Ratio
BT B/C	17106	$(85)^{200} = (13 + (4)^3 + ... + (4)^3)^{200}$	99.38%	44	388.77
SP B/C	26888	67^{400}	99.67%	89	302.11
CG B/C	41954	$(552)^{75} = ((21)^{26} + 6)^{75}$	98.68%	10	4195.4
MG B	8909	$(416)^{20}$	93.39%	590	15.1
MG C	10047	$(470)^{20}$	93.56%	648	15.5
LU B	203048	$(812)^{249} = ((4)^{100} + (4)^{100} + 12)^{249}$	99.58%	63	3222.98
LU C	323048	$(1292)^{249} = ((4)^{160} + (4)^{160} + 12)^{249}$	99.58%	63	5127.75
Average	71695		98.16%	165	1815.39

Table 4 focuses on the execution time for optimal compression. The total time for loop discovery is reported, which includes repeats discovery, loop identification and loop filtering. The largest and the smallest loop element sizes are also noted. One observation is that the repeats discovery time is a relatively small component of the total loop discovery time, which is dominated by the loop identification time. The loop filtering time was consistently very small in comparison and is not reported separately in the table.

The times for repeats discovery and loop discovery increase as the trace size increases. For example, class C LU benchmark takes 8028.83 seconds (2.23 hours) to finish discovering all possible repeats, and identification of loops from those repeats takes 113890.21 seconds (31.64 hours). As this is for a modest input data size running on only 16 nodes, the execution time is a major concern for realistic larger clusters and data sizes. Figure 2 plots the relationship between execution time and trace size.

Table 4 also shows the times at which the smallest loop and the largest loop was discovered during the compression of each benchmark trace. For all benchmarks, the largest loop was discov-

Table 4: Performance and execution time breakup for optimal compression. The loops discovery time includes the time for repeats discovery that is also listed separately, loop identification and loop filtering.

Name	Trace Length	Repeats Discovery Time (s) Total	Up to Smallest Repeat	Up to Largest Repeat	Loops Discovery Time (s) Total	Up to Smallest Element	Up to Largest Element	Loop Element Size Smallest Size	Largest Size
BT B/C	17106	12.85	0.47	0.87	311.18	0.63	4.84	4	85
SP B/C	26888	15.88	0.98	0.98	747.73	5.81	5.81	67	67
CG B/C	41954	239.29	1.46	5.78	2021.78	3.73	67.77	21	552
MG B	8909	35.85	0.00	3.95	113.48	0.00	13.74	1	416
MG C	10047	45.96	0.00	4.97	144.54	0.00	17.41	1	470
LU B	203048	2565.73	4.93	24.31	44204.82	6.51	463.18	4	812
LU C	323048	8028.83	7.83	59.72	113890.21	10.18	1172.63	4	1292

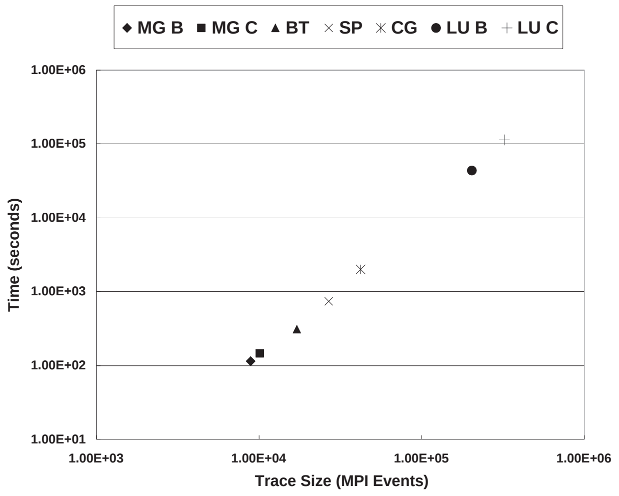

Figure 2: Time of loop discovery .

ered within a small fraction of time as compared to the total execution time. A similar pattern is observed for the largest repeats. Loop discovery with Crochemore's algorithm employs successive refinement from 1 up to half the trace size. However, the largest loop element in all cases was a small fraction of the trace size. Hence, the bulk of the time spent by the algorithm was after all the loops had already been discovered. Of course, in hindsight, the process could have been terminated earlier with the same results. However, there is no definitive way to be certain that optimal compression has been achieved, although heuristics can be developed based on the degree of compression already achieved.

The observations from the results of this optimal algorithm and the fact that it spent much of the time in processing that did not contribute to final compression was the motivation for us to develop a greedy compression algorithm. Greedy compression reduces the running time by reducing the length of the original string during compression as loops are discovered. Consider the trace of class C LU benchmark. Table 4 shows that the smallest loop contains 4 elements, so the reduction in trace size starts at level 4 by replacing those loops with new loop symbols. The largest loop has

1292 elements, which contains two inner loops with 4 elements iterating 160 times and other 12 elements, and the loops span 99.58% of the trace. Hence, after the next reduction, which happens at level 14, the trace size will be (100-99.58) = 0.42% of the original trace size, and compression will be virtually over. In contrast, optimal loop discovery procedure will have to execute with a trace size of 320348 until a level equal to half that size.

Table 5: Results of greedy compression

NPB Name	Trace Length	Total Time of Greedy Compression	Total Time of Optimal Compression	Major Loop Structure Discovered by Greedy Compression	Compression Ratio Optimal Algorithm	Compression Ratio Greedy Algorithm
BT B/C	17106	8.91	311.18	$(85)^{200} = (13 + (4)^3 + ... + (4)^3)^{200}$	388.77	388.77
SP B/C	26888	7.61	747.73	67^{400}	302.11	302.11
CG B/C	41954	8.48	2021.78	$\boxed{(552)^{75} = (5 + (21)^{25} + 22)^{75}}$	$\boxed{1353.35}$	4195.4
MG B	8909	8.64	113.48	$(416)^{20}$	15.1	15.1
MG C	10047	10.88	144.54	$(470)^{20}$	15.5	15.5
LU B	203048	33.16	44204.82	$(812)^{249} = ((4)^{100} + (4)^{100} + 12)^{249}$	3222.98	3222.98
LU C	323048	61.9	113890.21	$(1292)^{249} = ((4)^{160} + (4)^{160} + 12)^{249}$	5127.75	5127.75

The results from greedy compression are presented and compared with the optimal compression results in Table 5. The reduction in the execution time with the greedy approach is dramatic. The maximum compression time with the greedy procedure ranges from 7.6 seconds to 61.9 seconds, versus the range from 113 seconds to 113,000 seconds for the optimal procedure. Clearly, this is a much more promising approach for large traces. Since the greedy approach is not optimal, we report the loop nests discovered and the compression achieved for each benchmark. For 6 of the 7 benchmarks, the loop nests discovered and the compression achieved were identical with the optimal and greedy approaches. One exception was the CG benchmark as noted in Table 5. In this case the optimal procedure yielded a compressed trace size of 10, while the greedy procedure yielded a compressed trace size of 31. Clearly this is not a practical concern even though the degree of compression reported varies by a factor 3. The reason for the difference is clear when the loop nest discovered for the CG benchmark as shown in Table 5 is compared with the loop nest discovered as shown in Table 3. One discovered loop with 21 symbols is offset due to the greedy procedure such that it has one less iteration as compared to the optimal loop nest.

6 Conclusion

This paper has presented an efficient and practically optimal framework to identify complete loop nests from execution traces. The methodology constructs a loop nest from all repeating patterns identified by Crochemore's algorithm. A fast greedy approach is also developed. Experimental results with traces generated with Class B/C NAS benchmarks on 16 nodes demonstrate that the approach is effective for compression of communication traces. Both the optimal and greedy approaches discover similar loop nests and deliver similar compression results. However, the processing time rises to hours with the optimal procedure for modest length traces of 1000s of MPI calls. The greedy approach provides virtually identical compression at a fraction of the execution time of the optimal method. The maximum compression time for our test suite was around one minute with the greedy procedure. Most importantly, unlike many other compression heuristics, the greedy approach developed is theoretically proven to yield "near optimal" optimal results.

While many compression algorithms exists, the efficient discovery of long range repeating patterns due to outer loops in a trace is a significant challenge. To the best of our knowledge, this is the first effort to discover the optimal loop nest in execution traces. The procedure developed is general and can be applied to trace compression and similar problems in a variety of scenarios. We believe this is an important step forward in analyzing execution traces for performance modeling and performance prediction.

Acknowledgement: This material is based upon work supported by the National Science Foundation under Grant No. ACI- 0234328 and Grant No. CNS-0410797

References

[1] Alberto Apostolico and Stefano Lonardi. Some theory and practice of greedy off-line textual substitution. In *Data Compression Conference*, pages 119–128, 1998.

[2] Ravi P. Bhayankaram. Automatic identification of communication patterns in parallel applications. Master's thesis, University of Houston, May 2007.

[3] Maxime Crochemore. An optimal algorithm for computing the repetitions in a word. *Inf. Process. Lett.*, 12(5):244–250, 1981.

[4] Xiaofeng Gao, A. Snavely, and L. Carter. Path grammar guided trace compression and trace approximation. pages 57–68, June 19-23 2006.

[5] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[6] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all the tandem repeats in a string. *J. Comput. Syst. Sci.*, 69(4):525–546, 2004.

[7] Paul Havlak and Ken Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, 1991.

[8] J. Marathe, F. Mueller, T. Mohan, B. Supinski, S. A. McKee, and A. Yoo. Metric: Tracking down inefficiencies in the memory hierarchy via binary rewriting. In *CGO*, pages 289–300, 2003.

[9] Edward M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[10] C. Nevill-Manning, I. Witten, and D. Maulsby. Compression by induction of hierarchical grammars. In *Data Compression Conference*, pages 244–253, Snowbird, UT, 1994.

[11] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm, 1997.

[12] Craig G. Nevill-Manning and Ian H. Witten. Sequitur. `http://SEQUITUR.info`.

[13] M. Noeth, F. Mueller, M. Schulz, and B. de Supinskiin. Scalable compression and replay of communication traces in massively parallel environments. In *21th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2007)*, Long Beach, CA, April 2007.

[14] S. Sodhi and J. Subhlok. Automatic construction and evaluation of performance skeletons. In *19th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, CO, April 2005.

[15] Jens Stoye and Dan Gusfield. Simple and flexible detection of contiguous repeats using a suffix tree (preliminary version). In *CPM*, pages 140–152, 1998.

[16] A. Toomula and J. Subhlok. Replication memory behavior for performance prediction. In *LCR 2004: The 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston, TX, October 2004.

[17] Andrew Turpin and William F. Smyth. An approach to phrase selection for offline data compression. In *ACSC*, pages 267–273, 2002.

[18] Esko Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, 1995.

[19] Peter Weiner. Linear pattern matching algorithms. In *FOCS*, pages 1–11, 1973.

[20] Qiang Xu. *Automatic Construction of Coordinated Performance Skeletons*. PhD thesis, University of Houston, Aug 2007.

[21] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

A Results on near optimality of greedy compression

We first restate the main result about greedy trace compression informally:

The early reduction of all inner loops corresponding to a fixed loop body can impact the identification of a longer span outer loop only as follows: the body of the discovered loop may be a reordering of the body of the original loop, and the number of iterations in the discovered loop may be up to 2 fewer than the number of iterations in the original outer span loop.

We now present a set of results that will be employed to prove the above result formally. Recall that the notation (j, α, m) representing a PM-triple means that the corresponding PM-repeats start at location j in the string, the repeating substring is α and the number of repeats is m.

Lemma A.1 *Suppose PM-triple L represented as (j, α, m) is* leftmost *meaning that there are no PM-repeats of length $|\alpha|$ starting left of L, from location $i-1$. Let $A = |\alpha| - 1$ and assume $m > 2$. Then there exist PM-triples $(i+1, \beta_1, k_1)$, $(i+2, \beta_2, k_2)$, ...$(i+A, \beta_A, k_A)$ such that β_i is a rotation of α and k_i is either m or $m-1$.*

We refer to this group as the family of PM-triples corresponding to leftmost PM-triple L.

Proof. This result is stating the direct observation that, for every repeating sequence, starting with a forward offset smaller than the size of the repeating string yields another repeating sequence with at most one fewer number of repeats and with a rotated repeating string. □

Lemma A.2 *Let S be a string of symbols. Suppose there exist strings A and B such that:* $S = AB = BA$. *Then there must be another string C such that* $S = [C]^k$. *One implication is that* S *cannot be the repeating substring in a PM-repeat as it is not primal.*
Proof.

1. If $|A| = 1$, then it can be easily shown that S = $[A]^k$, where k = $|S|$. Same holds if $|B| = 1$.
2. If $|A| = |B|$, then S is of the form $[A]^2$.
   If either of the above cases holds, the result holds.
3. Otherwise, without loss of generality, let $|A| < |B|$. Then we can define a string T such that $B = TA$. Now, we have $S = AB = ATA$, and $S = BA = TAA$. This implies that $S = ATA = TAA$.

   We define $S = S'A$, with $S' = AT = TA$. We again have:

   (a) If $|T| = 1$, then $S' = [T]^{k'}$, where k' is $|S'|$. Hence, $S = [T]^k$, where k is $|S|$. Hence the result holds.
   (b) If $|T| = |A|$, then $S' = [T]^2 = [A]^2$, and $S = S'A = [A]^3$. Hence, the result holds.
   (c) Otherwise, S' can again be split as S was split in the previous level.

   The size of the string decreases by at least 1 in every new level. The result is proved based on the principle of induction.

□

Lemma A.3 *Given two PM-triples* L *and* S, *with repeating substrings* E_L *and* E_S, *respectively, where* $|E_L| > |E_S|$. *If there is an overlap between any interior instance (i.e. all instances except the first and the last) of* E_L *and the span of* S, *then the length of the span of* S *cannot equal or exceed* $|E_L|$.
Proof.
Without loss of generality, we assume that overlapping instances of E_L, and S are aligned at the left boundary as shown in Figure 3. If that is not the case, then the proof is generated with an aligned member of the family of PM-triples corresponding to PM-triple L as discussed in Lemma A.1.

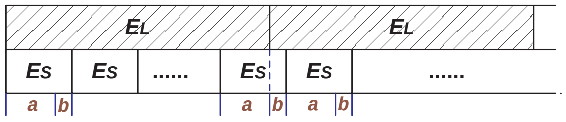

Figure 3: Overlapping PM repeats

If span of $S = E_L$, then clearly PM-triple S is not maximal as repeats of E_S continue in the next instance of E_L. Hence that cannot be true.
Suppose the length of span of $S > E_L$ in contradiction to the result to be proved. Then one instance of E_S will cross the boundary between instances of E_L as shown in Figure 3. (Since the first instance of E_L is an interior instance, a following instance must exist.) We split this instance

of E_S in two substrings a and b at the boundary as illustrated in Figure 3, i.e., $E_S = ab$. Now the left instance of E_L starts with E_S = ab, while the right instance of E_L starts with ba. But since they are repeats, they must be identical.

Hence we have $E_S = ab = ba$.

From Lemma A.2, $E_S = [x]^k$ for some x and k, which means that S is not a *primitive* repeat. Hence, by contradiction, the span of S cannot exceed E_L. Therefore, the span of S must be strictly smaller than $|E_L|$. □

We now present the main result formally.

Theorem A.4 *Consider PM-triple L represented as (j, α, m) with $|\alpha| > 2$. Let β be another substring with $|\alpha| > |\beta|$. Suppose every PM-triple with β as the repeating substring is identified and reduced to a symbol. As a result of these reductions, if L is not identified as a PM-triple, then another PM-triple L' (j', α', m') will be identified where,*

j' is between j and $j + |\alpha - 1|$,

α' and α are identical strings or one is a rotation of the elements of the other,

m' is between m and $m - 2$.

Proof.

Let S be a PM-triple with repeating substring β that overlaps with an interior instance of α corresponding to L. We initially assume that no other PM-triple with repeating substring β overlaps with this instance of α. From Lemma A.3 we know that the entire span of S must be smaller than $|\alpha|$. Further we assume for now that the entire span of S is contained within a single instance of α.

Under the above scenarios, every instance of α in L will contain S as part of the substring at the same location, which will be replaced by the same symbol. Hence the identification of the PM-triple corresponding to L will be unaffected by reductions of S.

Now suppose the span of S is *not* contained within a single instance of α and crosses two instances. In that case the above result can be proved for another member of the family of PM-triples corresponding to PM-triple L as discussed in Lemma A.1, although the number of repeats (or iterations) may be reduced by 1.

Finally, there can be multiple PM-triples with repeating substring β that overlap with the same instance of α. However these instances themselves cannot be overlapping - otherwise it can be shown that they are not PM-repeats based on Lemma A.2. The impact of non-overlapping PM-triples can be serialized leading to the same result as for a single overlapping PM-triple above.

The final result is that the number of repeats in the recognized PM-triples can be up to 2 less than L since none of the results applies to the first or the last instance of α in L. □