**6.895 Theory of Parallel Systems** Alexandru Caracaş

# Fast Serial-Append. Parallel File I/O Support for *Cilk*

**Project Report**

**Abstract**

We present a method for parallelizing serial applications strictly dependent on sequential file output. We give a scheme for maintaining *serial-append* for parallel programs. Subsequently, we describe the implementation of *serial-append* for the *Cilk* multithreaded language. Empirically, we show that a single file is a bottleneck in multithreaded computations that use files. We propose using multiple files to represent the same logical file and gain performance. Our results show that the application of concurrent skip lists, as a concurrent order maintenance data structure to support *serial-append*, scales well with the number for processors.

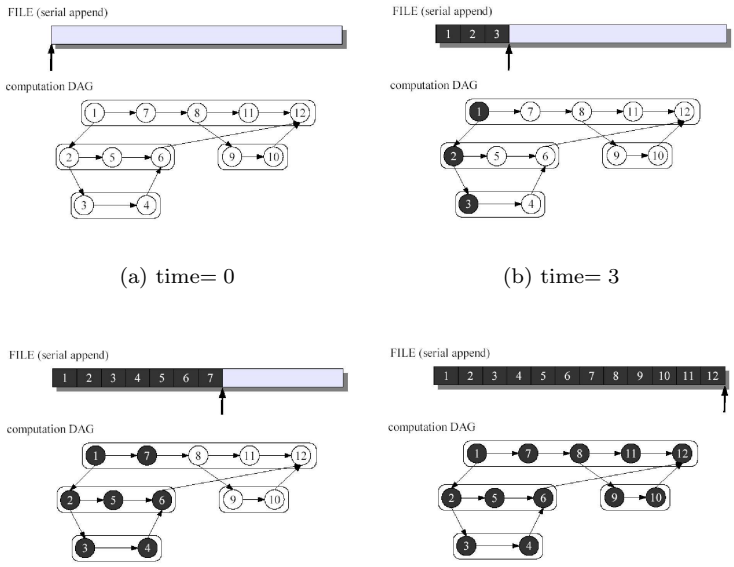
# Contents

We start by defining the concept of *serial-append* as a method of doing parallel file I/O. Next, we give a general idea for an algorithm to maintain *serial-append*. In the subsequent section, we discuss the implementation of *serial-append* for the *Cilk* multithreaded language. Following, we present the performance, experimental results of two implementation flavors.

# 1 Serial-Append

This section describes *serial-append* as a parallel file I/O mode. On most operating systems, *serial-append* is equivalent to opening a file for writing in append mode. This means that written data always gets appended to the end of the file.

We revise *serial-append* when executing sequentially, on a single processor. On a single processor, *serial-append* is equivalent with a depth first search traversal of the computation DAG. Consider we have a program, represented by the computation DAG from Figure 1, with threads numbered from 1 to 12. Consider that each thread in the computation DAG takes one unit time to execute. When the program gets executed we do a depth first search traversal and append all the written data from the threads at the end of the file. Figure 1 shows the *serial-append* file at different time steps during the single processor execution of the computation DAG. The already executed threads are shaded. A pointer represents the current position of the processor withing the file.

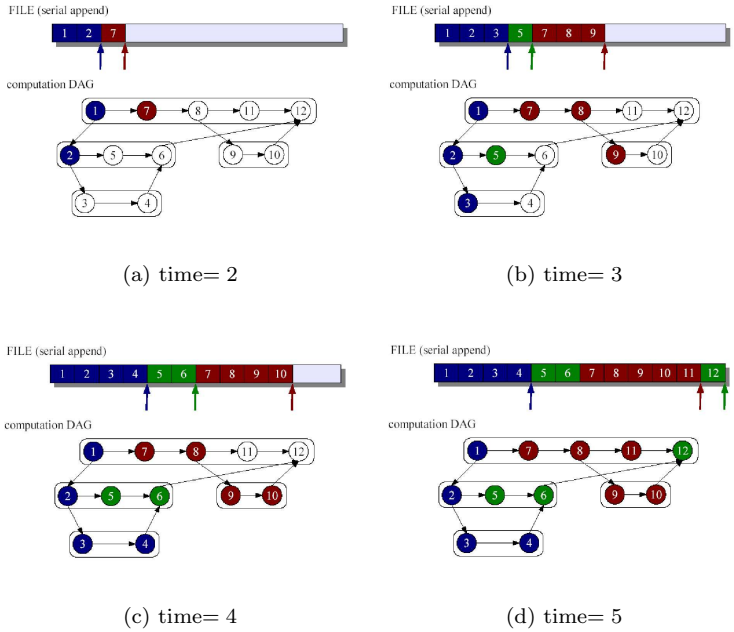



(a) time= 0 (b) time= 3

(c) time= 7 (d) time= 12

**Figure 1:** Different time-steps of the serial execution of a program that writes to a *serial-append* file.

Next, we describe the parallel case for *serial-append*, when the program is executed on multiple processors. On multiple processors, *serial-append* is equivalent to each processor having a pointer within the file and being able to insert data at that point. Consider the same program represented by computation DAG as in Figure 1, with the threads taking one unit of time to execute. Also, consider that we have three processors, with colors red, green, and blue, which execute the computation in parallel. Figure 2 shows the *serial-append* file at different time steps during the multiprocessor execution on the computation DAG. The already executed threads are colored based on the processor who executed them. Each processor has its own pointer that represents its position within the file.




(a) time= 2 (b) time= 3

(c) time= 4 (d) time= 5

**Figure 2:** Different time-steps of the multiprocessor execution of a program that writes to a *serial-append* file.

We define *serial-append* as a file I/O mode that allows for parallel execution with the same final file output as the sequential execution. From a conceptual stand point, *serial-append* from parallel applications requires an insert into file primitive. With such a file primitive the implementation of *serial-append* would be greatly simplified. Since file systems available today do not support such a primitive, the implementation of serial append for parallel applications is more challenging. The concept of *serial-append* is general and can be applied to any multithreading environment. In the following, we will discuss the implementation issues of *serial-append* for *Cilk*, a multithreaded language with a provably good work stealing scheduler.

## 1.1 Applications

A usual operation performed by many applications that use files is to append data to a file. Examples of such applications include compression utilities, logging programs, and databases. Since these applications are strictly dependent on the sequential output to the files they are using, parallelizing them is extremely challenging.

Ideally, we would want to execute these applications in parallel, and benefit from their eventual parallelism, improving their performance. A good example is the compression tool *bz2*, which is based on block compression, where each block compression is independent. Hence, we want to perform all write operations to a *serial-append* file in parallel, while still being able to have as output, conceptually, the same *serial-append* file as the sequential execution. Another important goal is to be able to read and seek efficiently within the *serial-append* file that was written in parallel.

Next we give a clear definition of the exact semantics of *serial-append* and the view of the programmer.

## 1.2 Semantics

The external API for *serial-append*, available to the *Cilk* programmer is similar to the file I/O API available on most operating systems. This similarity enables serial applications requiring sequential output to be parallelized easier using *serial-append*. The *serial-append* API includes the following functions:

- `open (FILE, mode) / close (FILE)`,
- `write (FILE, DATA, size)`,
- `read (FILE, BUFFER, size)`,
- `seek (FILE, offset, whence)`.

The semantics of *serial-append* are intuitive. The `write` operation, executed on a file opened in *serial-append* mode, maintains the correct ordering of the appends to the file, while executing in parallel. The `read` and `seek` operations can only occur after the file is closed, or on a newly opened file, otherwise we will have a data race situation. The semantics of `write`, `read`, and `seek` for *serial-append* are summarized in the following:

- **write** operations maintain the correct *serial-append* of the file, with the final output the same as the sequential, single processor, execution;
- **read** and **seek** operations can occur only after the *serial-append* file has been closed, or on a newly opened *serial-append* file.

Note: The `open` and `close` operation maintain the familiar semantics.

### 1.3 Related Work

This project leverages on *Cheerio*, previous work done by Matthew DeBergalis in his Master Thesis ([DeB00]). *Cheerio* includes a more general Parallel I/O API and implementation in the *Cilk* multithreaded language, and is described in [DeB00]. *Cheerio* provides a more general frame work and describes several possible schemes of performing file I/O from parallel programs, using global, local, or *serial-append* mode.

We focus on *serial-append* as a parallel file I/O mode and have two major differences with respect to the algorithm used in [DeB00] for performing *serial-append*. The first difference is that we use multiple files to allow each processor to write at will, unrestricted, whereas *Cheerio* uses a single file. As our experimental results show, using a single file on multiple processors is a major bottleneck in parallel computations that use that single file. The second difference is that we use a different data structure to keep the metadata about the execution of the parallel computation, namely a concurrent skip list, whereas *Cheerio* uses a data structure very similar to linked list.

## 2 Algorithm

### 2.1 General Algorithm Idea

We present a general algorithm idea used for maintaining the *serial-append* of a file from parallel programs. We reduce the problem of maintaining the *serial-append* of a file by braking the execution into parts executed by a processor and performing the appropriate bookkeeping and ordering of the parts.

**Observation 1 (serial-append)** ***Serial-append*** *is equivalent to a* **depth first search** *traversal of the computation DAG.*

Maintaining *serial-append* from a parallel program depends on the threading environment and the scheduling scheme used. Without file system support of a primitive to insert into a file, we need to do bookkeeping during the parallel execution of the program for maintaining the *serial-append*. Our focus will be on *Cilk* and its work stealing scheduler as described in [BFJ+96].

## 2.2 *Cilk* Algorithm for *serial-append*

We describe a *serial-append* algorithm for *Cilk*. *Cilk* is a multithreaded programming language with a provably good work-stealing scheduler. In *Cilk*, a spawn of a procedure is a little more than a function call and the processor that spawned the procedure immediately executes the spawned procedure. However, on a steal operation a processor goes and steals work, or procedures from a different processor. Hence, we have that a spawn preserves the depth first search traversal of the computation DAG. The problem arises with the steal operations which disrupt the normal depth first search traversal of the computation DAG.

**Observation 2 (steals)** *Steals affect the correct ordering for a* serial-append *file.*

Without a file system primitive that can support the insert into a file operation we need to do bookkeeping during the parallel execution of the program and account for the steal operations.

We simplify the problem of maintaining the *serial-append* of a file by partitioning the execution of the computation and properly maintaining the order of the partitions. A partition (or a PION) is represent by the following definition.

**Definition 3 (PION)** *A **PION** (**P**aralle**L** **I/O** **N**ode) represents all the write operations that a processor performs in-between two times it steals. A PION contains metadata, as well as a pointer to the actual data as we show in Figure 3.*

```
union MetaData{
        int victimID;             // the ID of the victim from which the PION was stolen
        unsigned long numbytes;   // number of written data bytes
        int fileid;               // the file where this node's data is written
}
```

**Figure 3:** Metadata fields contained by a PION.

Leveraging on the above definition, we give the algorithm for maintaining *serial-append* in *Cilk*.
**Algorithm:**

1. All PIONs are kept in an order maintenance data structure.
2. Every active processor has a current PION.
3. On each steal performed by a processor $P_i$ from processor $P_j$, with current PION $\pi_j$, we perform the following two steps:
   (a) we create a new, empty PION, $\pi_i$, and
   (b) attach $\pi_i$ immediately after $\pi_j$ in the order maintenance data structure.

The order maintenance structure, at a very basic, intuitive level, is represented by a linked list. The operations supported are: insert an element after a given element, search for an element with a certain rank, delete a given element, update an element; where an element is given by a pointer. For our case, the order maintenance data structure also needs support for concurrent operations.

# 3 Implementation

In this subsection we present the two implementations flavors of *serial-append* for *Cilk* and we discuss the issues with porting *Cheerio* to the new version of *Cilk*. We will refer to *serial-append* for *Cilk* as PLIO, standing for **P**aralle**L** File **I/O**.

PLIO interacts closely with the *Cilk* scheduler and runtime system and implements hooks on the steal operations. On each steal, the runtime system makes a call to internal PLIO routines, part of the runtime system, to do the required bookkeeping about the parallel execution. The external PLIO API available to the programmer also interacts with the *Cilk* runtime to perform the file I/O operations. The actual reading and writing from disk use the low level file I/O API provided by the operating system. The most significant work for PLIO was implementing the PLIO runtime as part of the *Cilk* runtime system together with the implementation of the external PLIO API.

Multiple, separate files are used as buffers for each processor to allow them to write to disk unobstructed. The metadata data structure is kept in memory, and written to disk when the file is closed.

## 3.1 Naïve Implementation

This subsection describes the first prototype of *serial-append* for *Cilk*. The naïve implementation uses a simple concurrent linked-list maintain the ordering of the metadata about the parallel execution of a *Cilk* program.

The PLIO implementation using linked lists for maintaining the order of PIONs represents the first implementation and it is simply a prototype. Because the data structure used to maintain the order is a concurrent linked list, the seek operations are extremely inefficient, namely they take $O(N)$ time, where $N$ is the number of PIONs (the size of the linked list). However, a simple concurrent linked list offers the advantage of very efficient inserts and update operations, in $O(1)$ time, together with much easier algorithms for inserting and deleting PIONs.

## 3.2 Improved Implementation

We describe the improvements to the very simple prototype, which involved using skip lists as un underlying concurrent data structure for order maintenance, as described in [Pug90]. Skip lists are a probabilistic variant to balanced trees and have logarithmic performance, namely all operations execute in $O(\log N)$ time, where $N$ is the size of the structure (in our case the number of PIONs).

In the improved implementation we use a variant of skip lists as un underlying data structure to maintain the order of the PIONs. The data structure being used can be defined as a rank order statistics double linked skip list. Figure 4 visually represents the data structure. Every node has one or more levels. Each level has forward and backward pointers to the immediate nodes with the same level. In this way, nodes with a higher number of levels can skip over the nodes with a low number of levels. There are no absolute keys stored in the nodes. Instead we use the metadata kept in the nodes, namely the number of data bytes written from this node to the *serial-append* file, to compute the absolute offset within the file. The computed absolute offset, represents the rank of a PION. The existent concurrent skip list algorithms were adapted accordingly to support backward pointers.

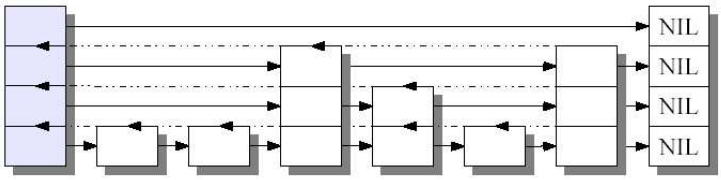



**Figure 4:** Visual representation of a double linked skip list.

When inserting, we give a pointer to the PION after which we want to insert. Using this data structure, we have logarithmic performance in the size of the structure, for all the insert, search, delete and update operations, namely $O(\log N)$, where $N$ is the size of the structure, in our case the number of PIONs.

### 3.3 Cheerio

In this subsection we describe the issues with porting *Cheerio*, written for *Cilk* 5.2, to the new version of *Cilk* 5.4. Porting *Cheerio* was a real challenge due to the fact that *Cilk* has been restructured substantially from the earlier versions. The *global* and *local* parallel file I/O modes from the *Cheerio* API, were ported successfully and are working well for the simple cases and sanity checks that were tested.

The port of *Cheerio* is not a complete success due to deadlocks issues with the implementation of the *serial-append* append mode. For documentation purpose, we give the description of the deadlock problem. During the program execution, two pointers, which should logically be distinct, become the same. Because there is a locking sequence involving acquiring locks for these particular pointers, the program deadlocks. More detail about the problem is documented in the code (in function `cio_commit_node`, see HTML API). The problem usually arises with the increase in the number of processors or execution time, and it manifests when committing nodes to disk. Due to time constraints, we were not able to remediate the problem in a timely manner to allow for a performance comparison with PLIO.

## 4 Performance

### 4.1 Testing Environment

The tests were performed on `yggdrasil.lcs.mit.edu` a 32 processor Origin machine running *Linux* (see Table 1 for the technical specifications).

| | yggdrasil: Origin 2000 w/R10000 |
|---|---|
| Operating System | IRIX64 6.5 07121148 IP27 mips |
| C Compiler | gcc - 3.2.2 |
| Processor speed | 195 MHz |
| Number of processors | 32 |
| Registers | 64 |
| L1 Cache size | 32KB |
| L1 Cache access time (ns) | 10 |
| L1 Cache line size | 128B |
| L1 Associativity | 2-way |
| L1 Cache lines | 256 |
| L2 Cache size | 4MB |
| L2 Cache access time (ns) | 62 |
| L2 Cache line size | 128B |
| L2 Associativity | 2-way |
| L2 Cache lines | 32K |
| Memory access time (ns) | 484 |

**Table 1:** Technical specification of testing environment.

As benchmark for the tests, we used a modified version of the example program `fib.cilk`, which computes the Fibonacci numbers. We chose this multithreaded program due to its high parallelism, approximately 300, measured with the *Cilk* profiling tools. The modified version of `fib` writes 10KB of data to a *serial-append* file for each recursive function call. It turns out that, if we want to compute the 25th Fibonacci number, we end up writing approximately 2GB of data to disk.

The running time for the two programs were measured with the `time` command, and the "real time" measurement was considered. The results of the tests are summarized in the following section.

## 4.2 Experimental results

In this subsection we present the experimental results and benchmarks for the two versions of PLIO. We also show an interesting experimental result related to scalability of parallel computations that want concurrent access to a single file. Based on the experimental result we argue that speedup can only be achieved when using multiple files.

### 4.2.1 PLIO

In this subsection we show the experimental results of the modified `fib` that writes to disk each time we do a function call. The difference in performance between the two implementation flavors of PLIO is practically indistinguishable when compared with the normal `fib` program.

Figure 5 compares the normal `fib` program with the modified `fib` program (represented by "PLIO" in Figure 5) that writes 2GB of data to disk in parallel. The graph shows linear speedup (execution time(seconds) times the number of processors) versus the number of processors. The scale is logarithmic since the execution time for the normal `fib` program is very small compared to the modified version which writes 2GB of data to disk. We notice that the normal `fib`, which performs no file I/O operations (represented by "NOIO" in Figure 5) achieves almost linear speedup. The same is true for the version of `fib` that performs disk I/O, which shows the good scalability of the *serial-append* scheme. On the same graph is plotted a simple C program (represented by "Pthread-sep"), described in the following subsection, which performs write operations to disk as fast as possible. The C program uses Pthreads and multiple files, as the implementation of PLIO in the *Cilk* runtime system. The C program emulates the bare disk operations of PLIO. The difference in performance between the Pthread bare emulation and PLIO is due to the *Cilk* overhead present in `fib`.

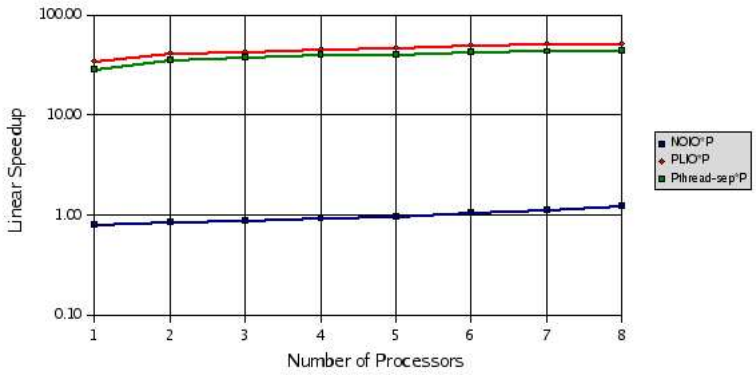



**Figure 5:** Performace of PLIO with respect to linear speedup.

The two implementation flavors of *serial-append* have almost the same performance for writing to disk. Figure 6 shows a comparison of the two PLIO flavors. The graph shows the execution time in seconds with respect to the number of processors. The linked list implementation is labeled as "PLIO" and the skip list implementation is denoted by "PLIO-SL".

Important to note is that the skip list implementation gains in performance, with respect to the linked list implementation, when reading and seeking (which involve search operations on the order maintenance data structure) from a serial append file. Also, from Figure 6, we see that write operations (which involve inserts into the order maintenance data structure) for the skip list implementation of PLIO are in practice as effective as the linked list implementation. The experimental results assert that skip lists are an efficient and simple alternative to more complicated concurrent order maintenance data structures, such as concurrent balanced trees.

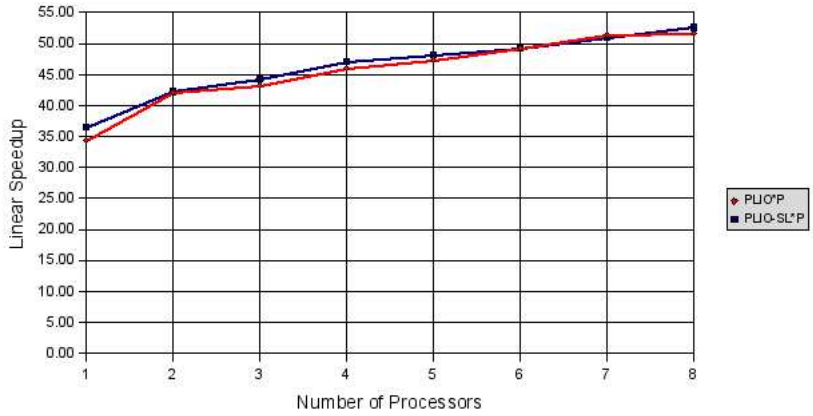



**Figure 6:** Comparison of the two implementation flavors (linked-lists and skip-lists) for *serial-append* .

### 4.2.2 Pthreads

We conducted a very simple experiment involving writing data to disk as fast as possible using a C program. We had three versions of the C program. The first version (represented as "FASTIO" in Figure 7) is a serial program that simply writes data in chunks of size 10KB . The second version (labeled as "Pthread" in Figure 7) uses a single file and multiple Pthreads to write in parallel to the file. The Pthreads seek to disjoint offsets within the file and start writing concurrently . The third version (labeled as "Pthread-sep" in Figure 7) also uses multiple Pthreads that write data in parallel. However, each Pthread has its own separate file.

The best results were obtained when using multiple Pthreads that write to separate files. The multiple Pthreads represent an emulation of what the PLIO runtime actually performs. When using a single file and multiple Pthreads that seek to separate offsets within the file and start writing in parallel, virtually, no speedup is possible. The results are summarized in Figure 7.

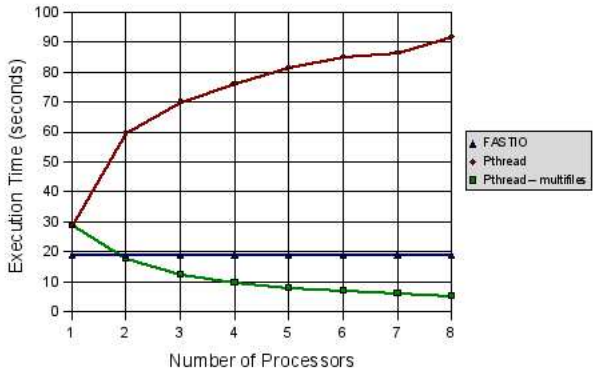



**Figure 7:** Performace of different versions of a C program that writes 2GB of data to disk as fast as possible.

## 4.3 Interpretation

The interpretation of the fact that, when using a single file, is impossible to achieve speedup, may come from the way in which the operating systems performs the file I/O operations. One possibility is that the

operating system uses internal kernel locks for each access to the file, literally serializing the multiple threads that are trying to write data to the same file.

A solution to this problem, as the empirical results show, is to use, instead of a single file, multiple files which represent a logical single file (as it is being implemented in PLIO). Another solution would be to have specific kernel and file system support for parallel access to files.

# 5 Additional Documentation

The implementations are available as follows:

- *Cheerio* - `https://bradley.csail.mit.edu/svn/repos/cilk/current_parallel_io`
- PLIO (with Linked List) - `https://bradley.csail.mit.edu/svn/repos/cilk/plio_append`
- PLIO (with Skip List) - `https://bradley.csail.mit.edu/svn/repos/cilk/plio_skip_list`

More detailed documentation about the C implementation and the API, individual methods and structures used by *serial-append* is available in HTML format in the `doc` folder for the respective implementation, as follows:

- *Cheerio*: `doc/cheerio`.
- PLIO (both flavors): `doc/plio`.

The file `index.html` is the entry point for the HTML documentation.

Difference files are available to ease the task of finding the pieces of code that were added to the current *Cilk* runtime-system. They are located inside the `doc` folder (together with the HTML API documentation) for the respective implementation (as described above), and have the extension `.diff` .

Additional documentation, available in the respective `doc` folders for the two implementations of PLIO, includes the final pdf presentation as presented in the *6.895 - Theory of Parallel Systems* class.

The programs used for the benchmarks and sanity checks are available in the `examples` folder for the respective implementation as follows:

- *Cheerio*: `examples/cheerio`.
- PLIO (both flavors): `examples/plio`.

# 6 Conclusion and Improvements

In this subsection we conclude our results and suggest several improvements for future work.

## 6.1 Summary of Results

We briefly summarize our results:

- Two flavors of *Cilk* runtime support for performing *serial-append*, with good scalability.
- Port of *Cheerio*, a more general parallel file I/O API, to *Cilk* 5.4.
- Experimental results concerning scalability constraints, related to the operating system internal file locks, when accessing the same file from multiple threads.
- Assertion of concurrent skip lists as an efficient concurrent data structure.

### 6.2 Improvements

Possible improvements to the presented schemes for *serial-append* include:

- Deleting PIONs that contain no data. This improvement decreases the number of PIONs kept in the order maintenance data structure that maintains the metadata about the parallel execution.
- Develop a cache oblivious algorithm for the skip list used to maintain the order of PIONs.
- Experiment with other concurrent order maintenance data structures such as B-trees.
- Develop a new file system with an insert primitive, and support from the operating systems for concurrent operations on a single file from multiple threads.

## References


[BFJ+96] Robert D. Blumofe, Matteo Frigo, Chrisopher F. Joerg, Charles E. Leiserson, and Keith H. Randall. An analysis of dag-consistent distributed shared-memory algorithms. In *Proceedings of the Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 297–308, Padua, Italy, June 1996.

[DeB00] Matthew S. DeBergalis. A parallel file I/O API for Cilk. Master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, June 2000.

[Pug90] William Pugh. Concurrent maintenance of skip lists. Technical Report CS-TR-2222, 1990.