An Efficient Programming Model for Memory-Intensive Recursive Algorithms using Parallel Disks

Vlad Slavici[†,*], Daniel Kunkle[‡], Gene Cooperman[†*], Stephen Linton[♯]
[†] Northeastern University, Boston, MA
[‡] Google Inc., New York
[♯] University of St. Andrews, St. Andrews, Scotland
vslav@ccs.neu.edu, kunkle@google.com, gene@ccs.neu.edu, sal@cs.st-andrews.ac.uk

ABSTRACT

In order to keep up with the demand for solutions to problems with ever-increasing data sets, both academia and industry have embraced commodity computer clusters with locally attached disks or SANs as an inexpensive alternative to supercomputers. With the advent of tools for parallel disks programming, such as MapReduce, STXXL and Roomy — that allow the developer to focus on higher-level algorithms — the programmer productivity for memory-intensive programs has increased many-fold. However, such parallel tools were primarily targeted at iterative programs.

We propose a programming model for migrating recursive RAM-based legacy algorithms to parallel disks. Many memory-intensive symbolic algebra algorithms are most easily expressed as recursive algorithms. In this case, the programming challenge is multiplied, since the developer must re-structure such an algorithm with two criteria in mind: converting a naturally recursive algorithm into an iterative algorithm, while simultaneously exposing any potential data parallelism (as needed for parallel disks).

This model alleviates the large effort going into the design phase of an external memory algorithm. Research in this area over the past 10 years has focused on per-problem solutions, without providing much insight into the connection between legacy algorithms and out-of-core algorithms. Our method shows how legacy algorithms employing recursion and non-streaming memory access can be more easily translated into efficient parallel disk-based algorithms.

We demonstrate the ideas on a largest computation of its kind: the determinization via subset construction and minimization of very large nondeterministic finite set automata (NFA). To our knowledge, this is the largest subset construction reported in the literature. Determinization for large NFA has long been a large computational hurdle in the study of permutation classes defined by token passing networks. The programming model was used to design and implement an efficient NFA determinization algorithm that solves the next stage in analyzing token passing networks representing two stacks in series.

*This work was partially supported by the National Science Foundation under Grant CCF 0916133.

1 Introduction

In the past 10 years, migrating traditional single-CPU computations to clusters or supercomputers has become important, as developers are faced with the task of providing solutions to problems handling increasingly larger data sets. The expense of supercomputers precludes their wider use. Hence, the majority of academic and commercial institutions have embraced commodity computer clusters as a less expensive alternative, albeit with less RAM than supercomputers.

To get the most benefit out of the use of commodity clusters, developers need to use the aggregate disks of the cluster in order to solve as large problems as possible. A typical 32-node commodity cluster might have an aggregate of 32 terabytes of disk, compared to an aggregate of only 512 GB of RAM, which is a consequence of disk being two orders of magnitude less expensive than RAM.

However, the modern out-of-core program developer still has a challenging task ahead: migrating traditional, RAM-based algorithms from the CPU to external memory on a per-algorithm/per-problem basis. This work addresses that challenge by introducing a general programming model and a method of migrating traditional recursive RAM-based algorithms to external memory programming.

Research Contributions There are two main research contributions in this paper.

- A general programming model and method for migrating *recursive* algorithms to parallel disk-based computing
- A general method of migrating random-acess (that is at the heart of most *RAM-based* algorithms for sequential programming) to parallel streaming access

Although there already exist many papers (e.g. see [27, 28] and their citations) written on a variety of parallel disk-based algorithms, the algorithms they describe appear quite different from their RAM-based counterparts, and seem to mostly cater to the expert, i.e. the researcher already familiar with parallel disk-based computing. Moreover, these already existing algorithms are providing little insight on how to tackle future problems not already described. The research presented here attempts to alleviate these obstacles, by providing a general theory of migrating recursive RAM-based algorithms to parallel disks.

There exist a number of tools for parallel disks programming — such as MapReduce [12], Roomy [17] or STXXL [7] — which allow the developer to focus primarily on the algorithm, while the runtime library takes care of the underlying details, such as working with the filesystem and sending data over the network. While all these tools increase productivity, they do not address the design task of transforming a RAM-based algorithm into a parallel disk-based one.

We propose a programming model that is independent of the programming language/extension used. Unlike previous approaches, which focused on a single programming language (MapReduce, STXXL, HaLoop [10]), the proposed programming model can be implemented on top of many tools for parallel disk-based computing, as described in Section 2.2.

As validation of this programming model, we implement examples in C, using the Roomy extension library. Roomy was chosen because it provides a broad range of primitive data structures including bit arrays, hash tables, unordered sets (`RoomyList`), and other data structures important for symbolic algebra. For particular programs, there exist extensions of the map-reduce platform (HaLoop [10], Pregel [21], and others) that are also general enough to serve as an implementation platform for the proposed programming model. The programming model tries to be agnostic about the choice of implementation platform.

The ideas of this model were motivated by the particular needs for recursion in symbolic algebra.

An important part of many symbolic computations is working with very large graph-like data structures, that only fit on parallel disks: groups [25], binary decision diagrams [19] and others. Very large finite state automata are useful in many areas, within symbolic computations and outside of it, as discussed in Section 3.

When working with the above mentioned large graph-like data structures, techniques similar to dynamic programming are often employed (for example, see [19, 25]). Migrating dynamic programming to parallel disks is described in Section 2.1 and in our parallel disk-based implementation of the 0-1 Knapsack problem in Section 4.

In the next few paragraphs we discuss possible applications in symbolic computation that are likely to benefit from the programming model proposed in this paper.

Possible Applications in Symbolic Computations In computational group theory, a hugely important example is the matrix recognition project. The matrix recognition project uses Aschbacher's classification of the subgroups of the general linear group into 9 categories [3]. For each of the non-simple classes, the project provides an algorithm to identify a normal subgroup. The matrix recognition algorithm then recursively calls itself to identify both the normal subgroup and the quotient of the original group over the given normal subgroup. Naturally, large groups lead to computations requiring large memory.

In the case of permutation groups, the classic algorithms for base and strong generating sets, and for partition backtrack (group intersection, centralizer, normalizer, etc.), both offer natural examples of recursion in which memory requirements can grow (especially for large base finite permutation groups).

Another important example of recursion lies in algorithms for a sparse representation of multivariate polynomials in *many* variables over a finite field. For example, in polynomial multiplication the natural recursion leads to multiplications of polynomials in fewer variables.

Given the wealth of topics, we chose the problem of NFA determinization (subset construction) for extremely large finite automata because of a long-standing challenge problem in this area (see "*Token Passing Networks*" in Section 3). One outcome of the work was to produce a DFA consisting of almost two billion states.

A second, smaller example is provided for dynamic programming and the knapsack problem (Section 4). This example is included to demonstrate the breadth of applicability of the programming model.

In the rest of this paper, Section 2 presents the general programming model and method for migrating recursive RAM-based algorithms to parallel disk-based algorithms, together with a discussion on how this method fits with tools such as Roomy, MapReduce and STXXL. Section 3 illustrates how the theoretical method is applied in practice by looking at the well-know recursive legacy algorithm for converting a non-deterministic finite automaton (NFA) to a deterministic finite automaton (DFA) via subset construction, followed by minimizing the DFA. The NFA to minimal DFA software is then validated experimentally by running it on very large inputs obtained from an application in the field of Token Passing Networks [4]. Section 4 presents a parallel disk-based implementation of the Knapsack problem. Section 5 presents related work.

2 A General Method for Migrating Recursive Algorithms from the CPU to Parallel Disks

A recursive algorithm starts out with the initial problem, and it generates sub-problems whose results are used to solve the initial, larger problem. The generated sub-problems form a callgraph, which is a directed acyclic graph (DAG). The graph has an initial node, called the root, which describes the problem (the level 0 sub-problem), and the edges point from a sub-problem to the smaller sub-problems it depends upon.

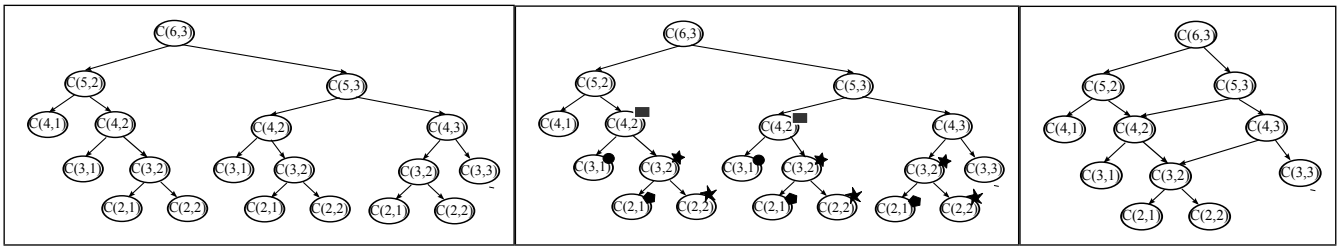
For migrating such a traditional recursive algorithm to parallel disks, we propose a general method based on traversing the problem callgraph.

Figure 1a presents the callgraph of a simple $\binom{n}{k}$ computation. Here $n = 6$ and $k = 3$. Even though there exists a simple formula for $\binom{n}{k}$, for the sake of explaining how the method works calculating $C(n,k) = \binom{n}{k}$ is here done by recursion:

$$C(n,k) = \binom{n}{k} = \begin{cases} \binom{n-1}{k} + \binom{n-1}{k-1}, n > 0, k > 0 \\ \binom{n}{1} = n, n > 0 \\ \binom{n}{0} = 1, n > 0 \\ \binom{n}{n} = 1, n > 0 \end{cases}$$

The $\binom{n}{k}$ algorithm is a simple, binary recursion. On the CPU, such an algorithm is straightforward to implement: most programming languages support recursive functions. Recursion is implemented by the programming language either by using the process stack, or by using a custom stack implementation. The implicit use of a stack leads to a depth-first exploration of the callgraph. Using the same approach for an out-of-core implementation is impractical, because depth-first search (DFS) relies heavily on random data access. It is well-known that streaming access should be used in a disk-based implementation instead of random access [25].

However, in order to solve the problem correctly, it is enough to explore its callgraph completely in any manner, not necessarily by DFS. For parallel disk-based programming, a breadth-first search (BFS) of the callgraph is more appropriate, since BFS can emphasize streaming access. Moreover, since a cluster has multiple nodes driving the locally-attached disks, a parallel version of BFS can be used (Parallel BFS) [20], which is the widest used pattern in computing with parallel disks.



(a) (b) (c)

Figure 1: *Simple recursive example:* $C(n,k) = \binom{n}{k}$. *(a) Callgraph for* $n = 6$ *and* $k = 3$. *(b) Highlighting overlapping sub-problems in callgraph. (c) Callgraph after merging overlapping sub-problems.*

The methodology proposed here is based on converting DFS to Parallel BFS. The important research contribution here is that we view the callgraph of a problem as an implicit graph to be explored. Whereas graph traversals are common in disk-based programming, they are performed on object graphs (the graph traversal itself is the solution - as is often the case in group enumerations [18, 24]). In our case, while the solution might be an object graph in some cases, the method is concerned with traversing the subject graph (a problem callgraph, which is not the answer to the problem, but a control and data flow graph). In the case of computing $\binom{n}{k}$, the answer is a number, so there is no object graph, but the method makes use of the traversal of the sub-problem callgraph.

Historically, the conversion of a complex binary recursive algorithm to one based on breadth-first search goes back at least to Ochi et al. [23], in their paper on the manipulation of very large binary decision diagrams (BDDs). Kunkle et al. [19] converted the algorithm further to one based on latency-tolerant, Parallel BFS, well-suited for parallel disks. However, at that time, the considered algorithm was not known to be just an instance of a more general method.

A generalization of binary recursive functions is the class of n-ary recursive functions (also called *exponentially* recursive) . Such a function makes n recursive calls to itself. This type of recursion corresponds to a callgraph in which a sub-problem can be solved by aggregating the results of n sub-sub-problems. Translating n-ary recursion to Parallel BFS can be summarized as follows:

- the first BFS frontier (represented as a list) contains an encoding of the problem
- for each sub-problem on the current BFS frontier, generate all n sub-sub-problems and add them to the next frontier.
- delayed duplicate detection is applied on the next frontier with respect to all previous frontiers, to see if any problems in the next frontier had already been discovered.
- duplicates are eliminated from the next frontier, but not before updating their parent nodes in the callgraph, so that parents point to the duplicates' representatives in the next frontier.
- after BFS finished (which happens when the next frontier is empty), scan each frontier, bottom-up, and for each node in the current frontier, merge information from the node's children.
- the algorithm finishes when the bottom-up scan reaches the root.

Some problems may allow representations in which nodes in the next frontier need not be compared to nodes from the previous frontiers [19]. In these problems, only sub-problems at the same BFS level can be duplicates of one another. The fewer frontiers that need to be checked at any one time during the program execution, the more efficient the program will be.

2.1 A Programming Model for Implementing Recursion on Parallel Disks

To further explain how recursive programs can be adapted into efficient parallel disk-based programs, we propose a programming model adapted from the Cilk [9] shared-memory model. In the Cilk model any task can create sub-tasks without blocking. These sub-tasks can be performed in parallel. The task (also called a predecessor) is complemented by a successor task, which does not start executing until all sub-tasks of the predecessor have completed. Except for the restriction placed on successors, any task can be executed concurrently with any other task. The programming model proposed here has many features in common with the original Cilk model, but important restrictions and additions were formulated so that it fits parallel secondary storage computations. These significant changes are:

- The directed acyclic graph (DAG) created by task dependencies is only explored by Parallel BFS. This means that the dependency callgraph has to be generated in a top-down fashion, and that, generally, a bottom-up scan of the successor tasks graph is necessary in order to execute the successors.
- Tasks will be generated and processed in batch, to alleviate the high-latency penalty of random accesses to disk.
- As opposed to the original Cilk model, in the adapted model sub-tasks cannot always be generated immediately. Some of the parameters needed to generate the sub-tasks reside on parallel disks, thus introducing the necessity of delayed batch access to obtain these parameters.

A graphical representation of the proposed programming model is presented in Figure 2.

Parallel BFS scans the predecessor sub-graph (generating the next BFS frontier at each step), while also generating the successor sub-graph frontier by frontier in batch, to avoid random access. As part of generating the predecessor sub-tasks, parameters might need to be accessed from parallel disks (denoted in Figure 2 by p_1 and p_2). These will also be accessed in batch. Once all predecessors have been generated, Parallel BFS is used to scan the successors bottom-up.

A single recursive step in the programming model for disk-based recursion is presented in Figure 3 and consists of 5 phases:

1. send a batch request to obtain necessary parameters (denoted by p_1, p_2, a.s.o) for generating the next BFS level of predecessor tasks from parallel disks.
2. receive a batched answer, containing p_1, p_2 a.s.o that were requested in phase 1.
3. use information encapsulated in the current level of predecessor tasks, together with parameters p_1, p_2, ... to generate the data and possibly the code for the next BFS level of predecessor tasks.

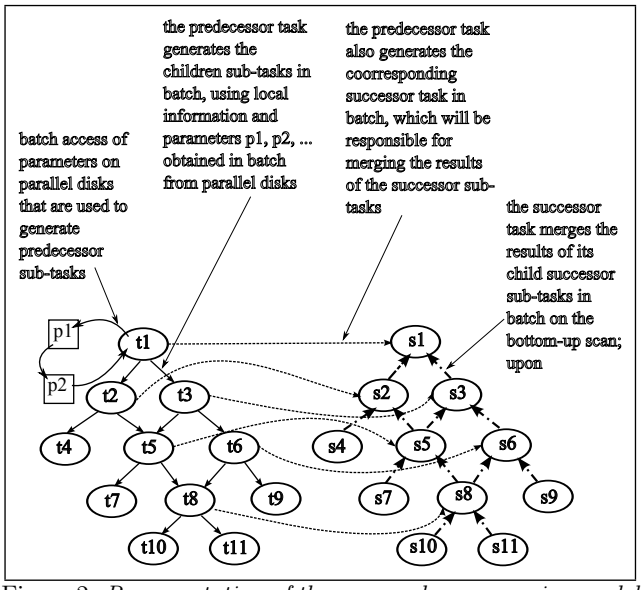


Figure 2: *Representation of the proposed programming model for adapting recursive algorithms to parallel disks. Tasks marked with t are top-down tasks (also called predecessors), while tasks marked with s are bottom-up tasks, also know as successors. Parameters already on parallel disks are denoted p_1, p_2 and there can be as many of them as necessary.*

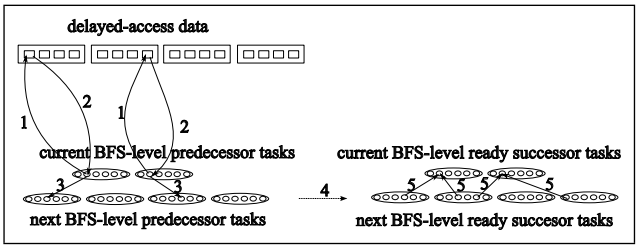


Figure 3: *Representation of one recursive step in the proposed disk-based programming model. One recursive step consists of 5 phases.*

4. generate the current BFS level of successor tasks - their purpose will be to aggregate the next BFS level of successor tasks, which has yet to be generated
5. connect successor tasks in current BFS level to their respective successor task parents, which have been generated in the previous recursive step.

Table 1 presents a brief comparison of the data structures and programming patterns generally used, at a high level, depending on the memory environment that the program data resides in. The table also shows which programming patterns are the most efficient for each of the possible memory environments.

Environ.	Algorithm	Data struct.	Duplicate detection	
			pointer equality	deep equality
RAM-based, sequential	recursion (DFS)	stack	memo. table	immediate (DFS)
	iteration (BFS)	queue		delayed
RAM-based, parallel	recursion (MT DFS)	per-thread stack	sync'ed memo. table	immediate
	iteration (MT BFS)	per-thread queue		delayed
disk-based, parallel	latency-tolerant Parallel BFS	latency-tolerant parallel queue	delayed duplicate detection	delayed, bottom-up scan

Table 1: *Fundamental algorithms, techniques and data structures for various environments: RAM-based sequential, RAM-based parallel, disk-based parallel. MT stands for multi-threaded. Memo. stands for memoization.*

Performance Enhancements This section addresses the problem of converting the performance enhancements of in-RAM data structures and algorithms to parallel disk-based programs.

A few general performance enhancements are used to increase the efficiency of RAM-based algorithms, the most important being memoization. Memoization generally means saving the results of already solved sub-problems. This way, if they are encountered again during the computation, their result is readily available.

On the CPU, the efficiency of memoization lies in the efficiency of in-RAM random access: looking up a stored result is fast. On parallel disks, the same technique is not applicable because of the high latency of a random data access. For parallel disk-based programs, this problem is solved by delayed duplicate detection [15, 16].

It is worth noting that memoization only helps with detecting sub-problems that are duplicates of previously generated sub-problems. Memoization cannot detect distinct sub-problems that have the same result.

Detecting equal sub-problems is done by pointer equality, while detecting distinct sub-problems with equal answers is done by deep value equality. For the latter, disk-based parallel programs need to perform an additional scan of the dependency graph, usually bottom-up, so that equivalent sub-problems can be detected (See Table 1).

Theoretical Performance of the Programming Model An n-ary recursion has branching factor n, but overlapping sub-problems account for a significant reduction in observed branching factor (this is the real branching factor r, often much smaller than n. For example, in the largest subset construction that we present in "*Experimental Results*" of Section 3, $n = 14$, but $r = 2.09$).

So the total number of generated predecessor tasks is $r^l \times n$, where l is the number of BFS levels. The number of generated successor tasks is at most r^l, which is also the size of the top-down callgraph.

In most data-intensive applications, in-RAM processing is much faster than disk access, so, for simplicity, we can ignore the time it takes to do CPU processing. For the most general case of duplicate detection new tasks have to be deduplicated against all previously discovered tasks. Assume we have a cluster with P nodes, D disk bandwidth and N newtork bandwidth. All tasks have to be written and read at least once from disk (in practice it could be more) and most of them will have to be sent over the network once. This brings the computation time to $\frac{k \cdot r^l \cdot (n+1)}{P \cdot min(D,N)} + \frac{j \cdot r^l}{P \cdot D}$. k and j are small constant factors which include the total data that needs to be read locally from disk without being sent over the network, such as in deduplication.

Another factor that one needs to be aware of is the synchronization time between BFS rounds, which depends on the cluster hardware and workload distribution.

2.2 Integrating the Programming Model with Practical Tools

This section describes how the proposed programming model for migrating recursive algorithms to parallel disks can be integrated with practical tools like Roomy, MapReduce or STXXL.

Integrating with Roomy Roomy [17] is a library for computing with parallel disks that is minimally invasive to the original, traditional algorithm. It provides a few high-level collection data structures (lists, arrays and hash tables) that the low-level runtime distributes to parallel disks, allowing the user to focus on the high-level algorithm.

Moreover, Roomy provides a smooth transition for the user from RAM-based random access to parallel disk-based streaming access. This is accomplished by allowing random access in the programming model, but having the runtime intelligently batching and organizing multiple random accesses so that they are presented to the file system in a delayed, streaming manner. Along with delayed random access, Roomy allows mapping a user-defined function over the elements of a collection.

Let's assume that we use a Roomy hash table to represent our current frontier in the callgraph, while a different hash table represents the next frontier, at this time empty. We can map a sub-problem generator function over the current frontier, which contains the current level sub-problems. The sub-problem generator function will first access, in a delayed batched random fashion, all additional parameters (p_1, p_2, ...) needed for computing the child sub-problems. Once these parameters have been accessed, the child sub-problems are computed and added to the next frontier. This corresponds to a batched recursive step.

The main recursive application presented in this paper (a package for converting an NFA to a minimal DFA - see Section 3) was implemented in Roomy, due to its expressiveness and the fact that it requires fewer changes to the subset construction algorithm, in comparison to MapReduce or STXXL.

Integrating with MapReduce MapReduce [12] is probably the most popular software for implementing parallel disk-based programs. Based on the idea of LISP loops, it provides two programming primitives: *map*, which allows mapping a user-defined function over large collections of data, and *reduce*, which aggregates the outputs of *map*, again by using a user-defined function.

MapReduce does not allow random access in the programming model. To the first-time user this might seem a major drawback, but it is offset by the ability of using *map* and *reduce* to define a *join* of two collections [8].*Join* is a term borrowed from databases, which basically means combining two collections of objects $A = \{a_1, a_2, ..., a_n\}$ and $B = \{b_1, b_2, ..., b_m\}$ into a collection of pairs $C = \{(a_{i1}, b_{j1}), (a_{i2}, b_{j2}), ..., (a_{ik}, b_{jk})\}$ based on some criterion. The *join* operation in MapReduce is equivalent to delayed, batched random access in Roomy.

In order to allow for efficient migration of RAM-based programs to parallel disks, in addition to *join*, one could build a *map-to-many* operation on top of *map* and *reduce* (*map-to-many* can be simulated by a simple *map* emitting a vector of elements).

HaLoop [10] is a notable extension of MapReduce in the sense that it provides built-in iterative support for programs, which is essential for migrating recursive programs to parallel disks.

The use of extension frameworks to MapReduce, such as HaLoop or the theoretical ones based on Datalog [1] offer support for a limited class of recursions (the ones that can be reduced to the Transitive Closure problem). Our proposed model also deals with recursions that cannot easily be converted to Transitive Closures. (See the dynamic programming application in Section 4).

Integrating with Pregel Pregel [21] is a framework for large-scale graph computations, which operates in *supersteps*. The user implements functions which specifiy the behavior of a vertex in a superstep. A vertex can receive messages sent to it in the previous superstep, its contents can be modified, and it can send messages to other vertices, that will be received in the next superstep.

Pregel was developed at Google to address the efficiency shortcomings of large graph processing in MapReduce. To quote from the Pregel paper [21]: *"MapReduce, however, is essentially functional, so expressing a graph algorithm as a chained MapReduce requires passing the entire state of the graph from one stage to the next—in general requiring much more communication and associated serialization overhead."*

Pregel passes data and control between vertices in synchronized rounds. In this regard it is close to Roomy, which passes data and control between elements in large data structures, also in synchronized rounds. Roomy and Pregel are probably, at present, the best fits for symbolic computations, due to their flexibility and expressiveness.

Integrating with STXXL STXXL [7] is an implementation of the C++ standard template library (STL) for external memory. While it supports multi-core parallelism, it does not support distributed memory (it is still under investigation), which makes it, for the moment, unsuitable for cluster computations. However, it supports multiple locally attached disks to the same compute node, which makes it a candidate for parallel disk-based computing. One has to keep in mind that there is usually a small practical limit to the number of disks that can be attached to the same machine, which is, in many cases, a severe limitation to the maximum size of the problems that can be approached using STXXL.

Since it is an extension of STL, it allows users to work with very familiar tools. However, users have to translate their algorithms into an asynchronous pipelining model. There is detailed information on how this can be done for various types of access patterns (diamond flow graphs, etc.). However, asynchronous pipelining is a much more restricted model compared to n-ary recursion.

STXXL could potentially be used to solve complex n-ary recursions like the NFA subset construction, which we implemented in Roomy and is presented in Section 3, but additional effort has to be invested into converting general n-ary callgraphs to asynchronous pipelined parallelism.

3 Application: From NFA to Minimal DFA

The first application of the proposed Programming Model is an algorithm for NFA determinization via subset construction. The implementation of the subset construction algorithm was validated on a series of challenge problems from the field of forbidden permutations.

Finite state automata (FSA) are usually the most computationally tractable form in which to analyze the regular languages that arise in many branches of computer science. That analysis requires efficient algorithms both for determinization of NFA (conversion of NFA to DFA) and minimization of DFA.

Recall that a *deterministic finite state automaton* (DFA) consists of a finite set of states with labelled, directed edges between pairs of states. The labels are drawn from an associated alphabet. For each state, there is at most one outgoing edge labelled by a given letter from the alphabet. So, a transition from a state dictated by a given letter is *deterministic*. There is an initial state and also certain of the states are called *accepting*. The DFA accepts a word if the letters of the word determine transitions from the initial state to an accepting state. The set of words accepted by a DFA is called a *language*.

A *non-deterministic finite state automaton* (NFA) is similar, except that there may be more than one outgoing edge with the same label for a given state. Hence, the transition dictated by the specified label is non-deterministic. The

NFA accepts a word if there exists a choice of transitions from the initial state to some accepting state.

Recall that the *subset construction* allows one to transform an NFA into a corresponding DFA that accepts the same words. Each state of the DFA is identified with a subset of the NFA states. Given a state A of the DFA and an edge with label α, the destination state B consists of a subset of all states of the NFA having an incoming edge labelled by α and a source state that is a member of the subset A.

Subset construction for large NFAs Subset construction can be viewed as a dynamic programming problem, which is traditionally implemented by recursion. Algorithm 1 describes the RAM-based recursive approach for subset construction.

Algorithm 1 RAM-based, Recursive Subset Construction

Input: Initial NFA, with initial state s_i and accepting states A_s
Output: DFA, represented as a list, equivalent to NFA

```
 1: Create subset I ← {s_i}, and an integer Id for it (Id_I)
 2: Insert pairs (I, Id_I) in a hash table of visited subsets, visited
 3: Create DFA — a list that will store the resulting DFA, containing 3-tuples of the form (Id, t, Id_next), meaning transition t takes state with Id to state with Id_next
 4:
 5: Call subset_construct(I, Id_I).
 6:
 7: func subset_construct(subset S, Id_S):
 8: for each NFA transition t do
 9:    Create empty subset S_next
10:    for each state s in subset S do
11:       t takes s to s_next in the NFA
12:       Add s_next to S_next
13:    if S_next is already in visited then
14:       Get Id_{S_next} from visited and add 3-tuple (Id_S, t, Id_{S_next}) to DFA
15:    else
16:       Create a new Id for S_next, add pair (S_next, Id_{S_next}) to visited and add (Id_S, t, Id_{S_next}) to DFA
17:       Recursively call subset_construct(S_next, Id_{S_next})
18: end func
```

Note that the recursive call from line 17 of Algorithm 1 is made for each newly discovered subset, which makes the branching factor of the recursion variable.

Using the method for migrating RAM-based, recursive programs to parallel disks from Section 2, the Programming Model of Section 2.1 and the performance enhancements (such as delayed duplicate detection) summarized in Table 1 of Section 2.1, we migrated Algorithm 1 to Algorithm 2, which we implemented in Roomy and ran on a computer cluster with 29 nodes. Note that *visited*, DFA and other data structures representing collections (*current_frontier*, *next_frontier*) are now parallel disk-based structures, in our case provided by Roomy. Experimental results which, to our knowledge, report the largest successully carried out subset construction compared to previous literature, are presented in Table 2 of Section 3.

In Algorithm 2, line 10 accounts for steps 1–3 in the Programing Model presented in Section 2.1. In this particular case, the parameters p_1, p_2, a.s.o. needed to generate the next BFS frontier are stored in RAM, since the NFA fits in memory. Line 11 accounts for the disk-based performance enhancement of delayed duplicate detection, presented in Table 1, which is equivalent to immediate in-RAM duplicate detection via a hash table. Line 13 implements step 4 in the Parallel Disk Recursive Programming Model - the creation of successor data that will contribute to the result.

Algorithm 2 Parallel Disk-based Subset Construction

Input: Initial NFA, with initial state s_i and accepting states A_s
Output: DFA, represented as a list, equivalent to NFA

```
 1: Create subset I ← {s_i}, and an integer Id for it (Id_I)
 2: Insert pairs (I, Id_I) in a hash table of visited subsets, visited
 3: Create DFA — a list that will store the resulting DFA, containing 3-tuples of the form (Id, t, Id_next), meaning transition t takes state with Id to state with Id_next
 4: Add pair (I, Id_I) into current_frontier and create next_frontier, initially empty.
 5:
 6: Call pardisk_subset_construct().
 7:
 8: func pardisk_subset_construct():
 9: while current_frontier is not empty do
10:    Call neigh(S, Id_S) for each pair (S, Id_S) in current_frontier, which returns a BATCH of S_next subsets.
11:    Perform delayed duplicate detection on the BATCH of S_next subsets with respect to visited and, for already visited subsets, obtain their Id, while for newly discovered subsets, create new Ids.
12:    Insert all new (subset S, Id_{subset S}) in visited.
13:    Add all (Id_S, t, Id_{S_next}) to DFA, for all t, regardless of whether S_next is new or not.
14:    Add all new S_next to next_frontier.
15:    Remove contents of current_frontier from parallel disks
16:    Rename next_frontier to current_frontier
17:    Create an empty list with name next_frontier
18: end func
19:
20: func neigh(subset S, Id_S) returns neighbor subsets:
21: Create ordered set of subsets SET, initially empty.
22: for each NFA transition t do
23:    Create empty subset S_next
24:    for each state s in subset S do
25:       t takes s to s_next in the NFA
26:       Add s_next to S_next
27:    Add S_next to SET.
28: Return SET.
29: end func
```

Note that, in this particular example, a bottom-up scan of successor data is not necessarily needed, since the answer is the successor data itself (the tuples of the DFA).

Finding the Unique Minimal DFA Besides providing an example of the proposed method for converting recursive, RAM-based programs to parallel disks, a secondary goal was to provide a Package for Very Large Finite Automata. In order for such a package to be useful in practice, the very large DFA obtained from subset construction needs to be minimized to a canonical form.

The algorithm chosen for computing the minimal DFA on parallel disks is based on a parallel RAM-based algorithm used on supercomputers in the late 1990s and early 2000s [14]. We call this the *forward refinement* algorithm. The central idea of the algorithm is to iteratively partition the states (to refine partitions of the states) of the given DFA, which is proven to converge to a stable set of partitions. Upon convergence, the set of partitions, together with the transitions between partitions, form a graph which is isomorphic to the minimal DFA. Initially, the DFA states are split into two partitions: the accepting states and the non-accepting states. For each state s, a 3-tuple is created: $(s, p, \{p_{next}\})$, in which p is the partition of s and $\{p_{next}\}$ is the ordered set of partitions of next states, obtained from s by following a fixed order of all transitions t. For each 3-

tuple, the pair $(p, \{p_{next}\})$ defines a new partition, in which s will be placed. The iterative process continues with the new set of partitions, until a refinement step yields no new partitions. Since this is an iterative algorithm working on constant amounts of data, it is easier to convert to parallel disks via a simple Parallel BFS. Experimental results for DFA minimization are provided in Section 3, Table 2.

Token Passing Networks A token passing network is a directed graph with designated input and output vertices. Numbered tokens are considered to enter the graph one at a time at the input vertex, and travel along edges in the appropriate direction. At most one token is permitted at any vertex at any time. The tokens leave the graph one at a time at the output vertex. A permutation $\pi \in S_n$ is called *achievable* for a given network if it is possible for tokens to enter in the order $1, \ldots, n$ and leave in the order $1\pi, \ldots, n\pi$.

The problem of achievable permutations by two stacks in series can be modelled as a finite token passing network and their behavior studied using the techniques of [5]. These techniques allow the classes of achievable permutations and the forbidden patterns that describe them to be encoded by regular languages and manipulated using finite state automata using a collection of GAP programs developed by the fourth author and M. Albert.

In previous work, the fourth author explored the cases of stacks of depths 2 and depth k for a range of values of k and observed that for large enough k the sets of minimal forbidden patterns appeared to converge to a set of just 20 of lengths between 5 and 9, which were later proved [13] to describe the case of a 2-stack and an infinite stack.

The application that motivates the calculations in this paper is a step towards extending this result to a 3-stack and an infinite stack, by way of the slightly simpler case of a 3-buffer (a data structure which can hold up to three items and output any of them).

Computations had been completed on various sequential computers for a 3-buffer and a k-stack for $k \leq 8$, but this was not sufficient to observe convergence. The examples considered in this paper are critical steps in the computations for $k = 9$, $k = 10$, $k = 11$ and $k = 12$. Based on the results of these computations we are now able to conjecture with some confidence a minimal set of 12,636 forbidden permutations for a 3-buffer and an infinite stack of lengths between 7 and 18.

Experimental Results Parallel disk-based computations were carried out on a 29-node computer cluster, each node's processor being a 4-core Intel Xeon CPU 5130 running at 2 GHz. Nodes on the cluster had either 8 or 16 GB of RAM and at least 200 GB of free disk storage and ran Red Hat Linux kernel version 2.6.9.

Table 2 presents the sizes of the intermediate DFAs produced by subset construction, the sizes of the minimal DFAs produced by the minimization process for the four considered token passing network problems, as well as the timings for both subset construction and DFA minimization.

4 Application: 0–1 Knapsack Problem

To help demonstate the generality of the approach presented here, we apply these methods to the knapsack problem, which is usually defined recursively and solved using dynamic programming.

Problem definition Given: n items, each with a positive weight w_i and positive value v_i; and a knapsack capacity W.

Maximize $\sum_{i=0}^{n} v_i x_i$ subject to $\sum_{i=0}^{n} w_i x_i \leq W$, where $x_i \in 0, 1$ represents whether item i is placed in the knapsack.

Recursive solution Define $M[i, w]$ as the maximum possible value when choosing from the first i items, given a knapsack of capacity w

$M[i, w]$ is defined recursively as

- $M[0, w] = M[i, 0] = 0$
- $M[i, w] = M[i - 1, w]$ if $w_i > w$
- $M[i, w] = max(M[i - 1, w], M[i - 1, w - w_i] + v_i$ if $w_i \leq w$

Parallel breadth-first algorithm M can be computed bottom up using dynamic programming, using the parallel breadth-first approach given in Algorithm 3.

Algorithm 3 Parallel breadth-first dynamic programming solution for the 0–1 knapsack problem.

Input: Item values $v_1, \ldots, v_n$, weights $w_1, \ldots, w_n$, and knapsack capacity W.

Output: Matrix M, where $M[i, w]$ is the maximum value achievable using the first i items and a knapsack of capacity w.

1: Set $M[0, w] = 0$, for $0 \leq w \leq W$.
2: **for** i = 1 to n **do**
3: **for** all $0 \leq w \leq W$ in parallel **do**
4: set $M[i, w] = \max(M[i-1, w], M[i-1, w-w_i]+v_i)$

Experimental results Algorithm 3 was implemented using Roomy [17]. It was tested for a variety of problems sizes using a shared-memory machine with with four quad-core 1.8 GHz AMD Opteron processors, 128 GB of RAM and 5 locally-attached disks adding up to 1.7 Terabytes, running Ubuntu SMP Linux 2.6.31-16-server, and compiling code with GCC 4.4.1

Items	Capacity	Size of M	Run-time (s)
16 K	16 K	256 M (2 GB)	1097
16 K	32 K	512 M (4 GB)	1668
16 K	64 K	1024 M (8 GB)	2724
16 K	128 K	2048 M (16 GB)	5000
32 K	16 K	512 M (4 GB)	2966
64 K	16 K	1024 M (8 GB)	9582
128 K	16 K	2048 M (16 GB)	31081

Table 3: Experimental results 0–1 knapsack problem.

Table 3 shows the running times for various size knapsack problems. The smallest example has 16 K items and a knapsack capacity of 16 K, resulting in a solution matrix with 256 M entries. This took 1097 seconds to complete.

The next three rows show the effects of increasing capacity, which adds additional columns to M. In this case, runtime increases sub-linearly as the syncronization overhead that occurs between computing each row is ammortized over the larger rows.

Finally, the last three rows show the effect of increasing the number of items. In this case, runtime increases superlinearly, as syncronization overhead is increased.

5 Related Work

Converting sequential programs to parallel ones has been an active topic of research for at least the past 25 years. The following two general results form the basis of the conversion from a sequential, RAM-based, recursive computation (which uses a stack) to a parallel, external memory, non-recursive computation (which uses the concept of a latency-tolerant Parallel Queue, described for the case of a binary decision diagram package in [19, Section 4.2]):

- *Primitive recursive functions can be converted into iterations* [6, 22]. Tail recursion can be converted into

Problem Instance	NFA size (#states)	Intermediate DFA size (#states)	Intermediate DFA Peak disk (GB)	Intermediate DFA time	Minimal DFA size (#states)	Minimal DFA Peak disk (GB)	Minimal DFA time
1	167,143	49,722,541	24	9min	32,561	6	38min
2	537,294	175,215,168	90	29min	95,647	22	2h 42min
3	1,667,428	587,547,014	327	3h 40min	274,752	81	9h 40min
4	5,035,742	1,899,715,733	1,136	1day 12h	774,172	295	1day 8h

Table 2: *Solutions for the four $NFA \rightarrow DFA \rightarrow minDFA$ considered problems.*

an iteration without using a queue. Binary or n-ary recursion can be converted to an iteration by replacing the process stack with an explicit stack (implemented using a queue).

- *Any program that uses the process stack could be modified to use the heap instead by employing continuation-passing style (CPS)* [26], which thus makes programs tail-recursive.

Significantly relevant to this research is the work of Arvind and Nikhil [2], who show that there is an equivalence between dataflow diagrams and CPS.

The idea of converting the process stack to a queue led to a successful shared-memory parallel language based on work-stealing: Cilk [9]. A task does not wait for sub-tasks to return, but instead it spawns a successor thread to receive the sub-tasks' results. The tasks are self-contained computations, also known as *closures*. To solve the problem that a task might need arguments that are not readily available, all missing arguments are treated as *continuations*.

Cormen and Davidson [11] proposed FG, a framework for generating efficient secondary storage cluster computations. Their research aims at hiding the effects of secondary storage high-latency accesses for computations whose data flow fits a pipeline structure, and later extended it for computations that fit fork-join patterns and directed acyclic graph (DAG) patterns. The dataflow paths in the DAG are determined dynamically, but the DAG structure is fixed and cannot be modified dynamically by the program. STXXL [7] shares many ideas with FG.

There has been significant work recently on implementing some classes of recursive algorithms on parallel disks [1, 10, 21], but these approaches are less general, and usually tied to programming in a specific programming language/extension.

6 References

[1] F. N. Afrati, V. Borkar, M. Carey, N. Polyzotis, and J. D. Ullman. Map-reduce extensions and recursive queries. In *Proc. 14th Intl. Conf. on Extending Database Technology*, EDBT/ICDT '11. ACM.

[2] K. Arvind and R. S. Nikhil. Executing a program on the MIT tagged-token dataflow architecture. *IEEE Trans. Comput.*, 39:300–318, March 1990.

[3] M. Aschbacher. On the maximal subgroups of the finite classical groups. *Inventiones Mathematicae*, 76:469–514, 1984. 10.1007/BF01388470.

[4] M. D. Atkinson. Generalized stack permutations. *Comb. Probab. Comput.*, 7:239–246, September 1998.

[5] M. D. Atkinson, M. J. Livesey, and D. Tulley. Permutations generated by token passing in graphs. *Theor. Comput. Sci.*, 178:103–118, May 1997.

[6] P. Axt. Iteration of primitive recursion. *Mathematical Logic Quarterly*, 11(3):253–255, 1965.

[7] A. Beckmann, R. Dementiev, and J. Singlery. Building a parallel pipelined external memory algorithm library. In *Proc. 2009 IEEE Intl. Symp. on Par.&Dist. Processing*, IPDPS '09. IEEE Computer Society.

[8] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in MapReduce. In *Proc. 2010 Intl. Conf. on Management of data*, SIGMOD '10. ACM.

[9] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37:55–69, August 1996.

[10] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. HaLoop: efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3:285–296, Sept. 2010.

[11] T. H. Cormen and E. R. Davidson. FG: A framework generator for hiding latency in parallel programs running on clusters. In *ISCA PDCS 2004*.

[12] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. In *Proc. 6th Symp. on Operating Systems Design & Impl. - Volume 6*. USENIX Association, 2004.

[13] M. Elder. Permutations generated by a stack of depth 2 and an infinite stack in series. *Electron. J. Combin.*, 13(1), 2006.

[14] J. F. JáJá and K. W. Ryu. An efficient parallel algorithm for the single function coarsest partition problem. In *Proc. Fifth Annual ACM Symp. on Par. Alg. and Architectures*, SPAA '93. ACM.

[15] R. E. Korf. Best-first frontier search with delayed duplicate detection. In *Proc. 19th Natl. Conf. on AI*, AAAI'04. AAAI Press.

[16] R. E. Korf. Delayed duplicate detection: extended abstract. In *Proc. 18th Intl. Joint Conf. on AI*. Morgan Kaufmann Publishers Inc., 2003.

[17] D. Kunkle. Roomy: A C/C++ library for parallel disk-based computation, 2010. `http://roomy.sourceforge.net/`.

[18] D. Kunkle and G. Cooperman. Twenty-six moves suffice for Rubik's cube. In *Proc. 2007 Intl. Symp. on Symb. and Algebraic Computation*, ISSAC '07. ACM.

[19] D. Kunkle, V. Slavici, and G. Cooperman. Parallel disk-based computation for large, monolithic binary decision diagrams. In *Proc. 4th Intl. Workshop on Par. and Symb. Comput.*, PASCO '10. ACM.

[20] C. E. Leiserson and T. B. Schardl. A work-efficient parallel breadth-first search algorithm (or how to cope with the nondeterminism of reducers). In *Proc. 22nd ACM Symp. on Par. in Alg. and Arch.*, SPAA '10. ACM.

[21] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. Pregel: a system for large-scale graph processing. In *Proc. 2010 Intl. Conf. on Manag. of data*, SIGMOD '10. ACM.

[22] A. R. Meyer and D. M. Ritchie. The complexity of loop programs. In *Proceedings of the 1967 22nd National Conference*, ACM '67. ACM.

[23] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. 1993 IEEE/ACM Intl. Conf. on Computer-aided Design*, ICCAD '93. IEEE Computer Society Press.

[24] E. Robinson. *Large implicit state space enumeration: overcoming memory and disk limitations*. PhD thesis, Boston, MA, USA, 2008. Adviser-Cooperman, Gene.

[25] E. Robinson, D. Kunkle, and G. Cooperman. A comparative analysis of parallel disk-based methods for enumerating implicit graphs. In *Proc. 2007 Intl. Workshop on Par. Symb. Comput.*, PASCO '07. ACM.

[26] G. J. Sussman and G. L. Guy L Steele, Jr. Scheme: An interpreter for extended lambda calculus. In *Memo 349, MIT AI Lab*, 1975.

[27] J. S. Vitter. External memory algorithms and data structures: dealing with massive data. *ACM Comput. Surv.*, 33:209–271, June 2001.

[28] J. S. Vitter and E. A. Shriver. Algorithms for parallel memory I: Two-level memories. Technical report, Providence, RI, USA, 1992.