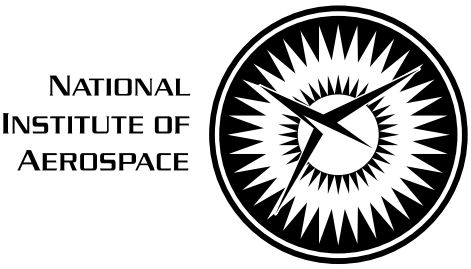
NASA/CR-2007-214543
NIA Report No. 2007-01



A Parallel Saturation Algorithm on Shared Memory Architectures

Jonathan Ezekiel and Gerald Luettgen
University of York, York, United Kingdom

Radu I. Siminiceanu
National Institute of Aerospace (NIA), Hampton, Virginia

February 2007

The NASA STI Program Office . . . in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA counterpart of peer-reviewed formal professional papers, but having less stringent limitations on manuscript length and extent of graphic presentations.

- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.

- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.

- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or co-sponsored by NASA.

- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and missions, often concerned with subjects having substantial public interest.

- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results ... even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at ***http://www.sti.nasa.gov***

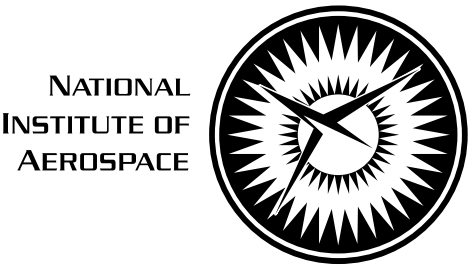
- E-mail your question via the Internet to help@sti.nasa.gov

- Fax your question to the NASA STI Help Desk at (301) 621-0134

- Phone the NASA STI Help Desk at (301) 621-0390

- Write to:
  NASA STI Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076-1320

NASA/CR-2007-214543
NIA Report No. 2007-01



A Parallel Saturation Algorithm on Shared Memory Architectures

Jonathan Ezekiel and Gerald Luettgen
University of York, York, United Kingdom

Radu I. Siminiceanu
National Institute of Aerospace (NIA), Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

Prepared for Langley Research Center
under Cooperative Agreement NCC1-02043

February 2007

Available from:

NASA Center for AeroSpace Information (CASI)
7115 Standard Drive
Hanover, MD 21076-1320
(301) 621-0390

National Technical Information Service (NTIS)
5285 Port Royal Road
Springfield, VA 22161-2171
(703) 605-6000

A PARALLEL SATURATION ALGORITHM ON SHARED MEMORY ARCHITECTURES

Jonathan Ezekiel*, Gerald Lüttgen†, Radu I. Siminiceanu‡

ABSTRACT

Symbolic state-space generators are notoriously hard to parallelize. However, the Saturation algorithm implemented in the SMART verification tool differs from other sequential symbolic state-space generators in that it exploits the locality of firing events in asynchronous system models.

This paper explores whether event locality can be utilized to efficiently parallelize Saturation on shared-memory architectures. Conceptually, we propose to parallelize the firing of events within a decision diagram node, which is technically realized via a thread pool. We discuss the challenges involved in our parallel design and conduct experimental studies on its prototypical implementation. On a dual-processor dual-core PC, our studies show speed-ups for several example models, e.g., of up to 50% for a Kanban model, when compared to running our algorithm only on a single core.

1 INTRODUCTION

Automated verification, such as temporal-logic model checking [8], relies on efficient algorithms for computing state spaces of complex system models. To avoid the well-known state-space explosion problem, symbolic algorithms working on *decision diagrams*, usually binary decision diagrams (BDD), have proved successful in practice [7, 16]. Several efforts have been made to implement these algorithms on parallel computer platforms, most notably on networks of workstations and on PC clusters [11, 12, 13, 17, 19]. The efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [17, 19] to sophisticated approaches relying on *slicing* BDDs [12], and techniques for *workstealing* [11]. However, the resulting implementations show only very limited speed-ups, which is not surprising given that state-space generation is essentially an irregular task.

Saturation [5], as implemented in the verification tool SMART [3], is a symbolic state-space generation algorithm with unique features (cf. Sec. 2). It is intended for asynchronous system models that are based on an interleaving semantics, and exploits the local effect of firing events on state vectors by locally manipulating multi-valued decision diagrams (MDDs) [15], which are a generalized version of BDDs. The algorithm has proved orders of magnitude more time- and memory-efficient than other symbolic algorithms [5, 6], including the one implemented in the popular NuSMV model checker [7]. Hence, the question arises whether the locality of events can also be utilized for parallelizing Saturation in order to

*University of York, York, Y010 5DD, UK Email: jezekiel@cs.york.ac.uk

†University of York, York, Y010 5DD, UK Email: luettgen@cs.york.ac.uk

‡National Institute of Aerospace (NIA), 100 Exploration Way, Hampton VA, 23666. E-mail: radu@nianet.org

This work was supported in part by the National Aeronautics and Space Administration under the cooperative agreement NCC-1-02043.

achieve further speed-ups. Previous approaches to parallelizing Saturation have focused on data parallelism [1, 2], but not on parallelizing the algorithm itself.

This paper investigates the parallelizability of the Saturation algorithm for shared-memory architectures, such as multi-processor, multi-core PCs. At first sight, this is a challenging endeavor since Saturation relies on relatively light-weight operations for "saturating" MDD nodes. Indeed, the algorithm's key operation is firing an event from a given MDD node. However, this operation is not an entity that can easily be parallelized, since newly generated nodes are saturated themselves before the saturation of the node under consideration continues. In this sense, Saturation is a greedy algorithm.

Almost two years of studies have given us a detailed understanding on what is needed to efficiently parallelize the firing of events within Saturation (cf. Sec. 3). Key to the algorithm is how to manage the dependency of tasks without forcing computation threads to frequently idle. To this end, we propose a task queue for storing tasks that need to be processed, from which available compute nodes can pick jobs. However, letting the operating system manage tasks is very costly, due to the overheads involved when creating operating system threads. Consequently, we implement our own implementation of thread pool that minimizes these overheads. Another challenge is how to group firings of events such that our tasks, while still being lightweight, become sizable. Our solution here is to consider firing several events for a given node within the same task.

We have implemented our algorithm on a PC with two dual core Intel processors. Our experimental studies (cf. Sec. 4) show speed-ups of up to 50% when running the parallel algorithm on four processors instead of one, for large systems with densely connected MDD nodes, such as a Kanban model. Indeed, the algorithm's efficiency depends on the studied models. Improvements over the heavily optimized sequential version of Saturation have proved to be hard to achieve. We carefully justify our results with the help of Intel Threading Tools [`www.intel.com/software/products/threading/`], which provides valuable insights into the locking behavior and processor idle times of our algorithm. The analysis also shows that our parallelization is quite efficient in terms of utilizing computation resources. However, speedups over the sequential algorithm are model dependent.

2 SATURATION

The introduction of Binary Decision Diagrams (BDD) has revolutionized the field of model checking. BDDs offer a compact encoding for large sets of states when performing the next-state computation in a single, symbolic operation. However, despite the clear advantage over explicit exploration algorithms, the traditional (monolithic BDD) approach has been inherently a breadth-first search (BFS) strategy, or a variant of it. For complex systems however, the symbolic BFS algorithm usually suffers from an excessive peak memory consumption, thus fails to build the entire state-space even when the final BDD is much smaller than the (intermediate) peak.

The Saturation algorithm is radically different from its symbolic predecessors. It consists of a series of small, nested fixed point operations that are guided by the current shape of the decision diagram with the goal of systematically *saturating* MDD nodes, in a bottom up fashion. The building block of this strategy is the firing of an individual event in an individual node (that encodes a subset of states), in contrast to computing the entire next-state function on the entire current set of states. This finer-grain decomposition of symbolic operations

is more flexible, by allowing more efficient firing orders, while exploiting the *event locality* property, which is inherently present in concurrent, asynchronous systems. In our setting, the system is *structured* if it consists of a collection of subsystems, such that the global system state can be written as a vector of *local* states and the effect of an event on the system state can be expressed as the composition of the local effects of the event on each subsystem. For structured systems, an encoding of sets of states with Multi-way Decision Diagrams (MDDs) is more natural, for several reasons. Firstly, the one-to-one correspondence between a state variable and the level in the MDD is always apparent. Secondly, MDD nodes allow and better exploit the key operation of *in-place-updates* ([4]) that accelerates the exploration.

A node is said to be *saturated* if it encodes a local fixed point (with respect to the subset of events that affect its level and the ones below). The global fixed point strategy is therefore *chaotic*, which can be shown to be correct as long as the firing order of events is *fair*, i.e. each event if considered often enough. To allow chaotic exploration, the system's model requires a disjunctively partitioned transition relation and to satisfy the interleaving semantics of firing events.

In contrast to traditional, BFS-oriented approaches, Saturation is extremely efficient. It performs lightweight decision diagram manipulations in contrast to the heavyweight, monolithic image computation of its symbolic counterparts. The greedy strategy of saturating every node immediately upon its creation (by pre-empting the undergoing event firing operation) results in a non-trivial series of recursive, preemptive firings, but also in a dramatic reduction of the peak MDD size. The intuition behind this is that *only* saturated nodes can be part of the state-space representation, while non-saturated nodes are guaranteed not to. Also, once a node is saturated, it does not need to be considered for further exploration. The bottom-up order of saturating nodes ensures that all descendants are already saturated when a node is considered for saturation. Last but not least, since the complexity of symbolic algorithms is closely related to the number of nodes in the decision diagram (as opposed to the number of encoded states), Saturation is significantly more efficient than BFS: up to several orders of magnitude faster and memory efficient on classic asynchronous systems [5].

Paradoxically, the properties of the Saturation algorithm makes the mission of effectively parallelizing it extremely difficult. Given the doubly-recursive dependencies of the saturation and event firing routines, the algorithm is sequential by nature and heavily optimized, leaving very little room for further improvements.

3 PARALLEL SATURATION

State space generation algorithms are difficult to parallelize due to the characteristics of the process. Tasks such as applying the next state function are irregularly sized, dependent upon each other and have to synchronize frequently. These characteristics can introduce high parallel overheads. Irregular tasks cause load imbalance, and dependencies between tasks compound the problem. Frequent synchronization introduces high synchronization overheads. While there are a number of techniques to load balance irregular tasks, frequent synchronization can only be avoided by making tasks as large and independent as possible. We exploit event locality to achieve this. Creating parallel tasks from event firings allows parts of the MDD to construct independently since subsequent event firings are local to the resultant sub-MDD.

While we can exploit event locality to create independent tasks, local events often cannot

be parallelized due to their efficiency. On the shared memory architectures we investigated, the cost of creating a parallel event to perform an in-place-update outweighs the cost of performing it. On a SPARC Solaris shared memory machine we approximated the cost of an in-place-update as 1200ns compared to 90000ns for creating a thread or 8000ns for allocating a task to an existing thread. An in-place-update can also occur when an event firing fully utilizes previous work that has been cached. The cost of retrieving information from a cache is only 900ns. We therefore group event firings together and only consider event firings that do not result in in-place-updates for parallel tasks.

To address irregularity we introduce a task queue to which parallel tasks can be added to. To load balance the tasks we utilize a thread pool where a thread is mapped 1-to-1 to a processor core. An available thread picks a task from the queue and performs the work associated with it. Fitting the Saturation algorithm into this load balancing structure is difficult due to its mutually recursive nature. In order to prevent threads from suspending we have to eliminate sequential waits on the result of a parallel event firing. We achieve this by introducing upward arcs into the MDD structure. Upward arcs directly replace recursive function calls waiting for work to complete. Instead the function calls continue when parallel work is pending leaving the upward arcs to represent future updates on a node. They allow a thread that was created by firing an event on a node to continue the work on the node when it completes its task. Each node must keep track of the number of threads operating on it in order to determine when it has reached a fixpoint. We allow work requests to be cached before they have been carried out to avoid duplicate work in parallel. A thread requesting uncompleted work can connect the thread carrying out the work to the node it is required for via an upward arc.

3.1 Algorithm Description

The result of mapping our ideas into code is shown in Fig. 1, with supporting functions described in Fig. 2. We extend the code from the sequential version in [5]. Parallel code is highlighted. Dark code facilitate tasks and removes mutual recursion. Light code shows locks ensuring correct synchronization. We denote a node using the notation $\langle k.p \rangle$ where k is the MDD level of the node and p is the unique index of the node. $\langle k.p \rangle[i]$ represents a downward arc from state i. Information and locks on the node are denoted by $\langle k.p \rangle.i/l$ where i/l is the information or lock. The MDD nodes are stored in a hash table, we call a *unique table* UT which stores nodes at each level i.e., $UT[k]$, $1 < k \leq K$ where K is the height of the MDD. Work resulting from event firings is stored on a per level basis in a cache we call a *firing cache* FC, i.e., $FC[k]$, $1 < k \leq K$.

Node Information: The node keeps track of the number of threads that are currently working on it or will perform work on it in the future (via upward arcs). The functions *AddOp* and *RemoveOp* allow current/pending thread operations to be added and removed from the node. The saturation status of the node is indicated by $\langle k.p \rangle.saturating$ and determines if a node with no operations is *saturated* from firing all events, or a newly created node waiting to be saturated. Nodes created from event firings store a *key* to add to the firing cache.

Initialization: The function *Gen* creates an initial MDD and threads. Each thread calls *ThreadLoop* to synchronize on the task queue. Tasks are added to the queue for the bottom nodes of the MDD.

Saturate: indicates that the node has begun saturating by setting *saturating* to true. Since

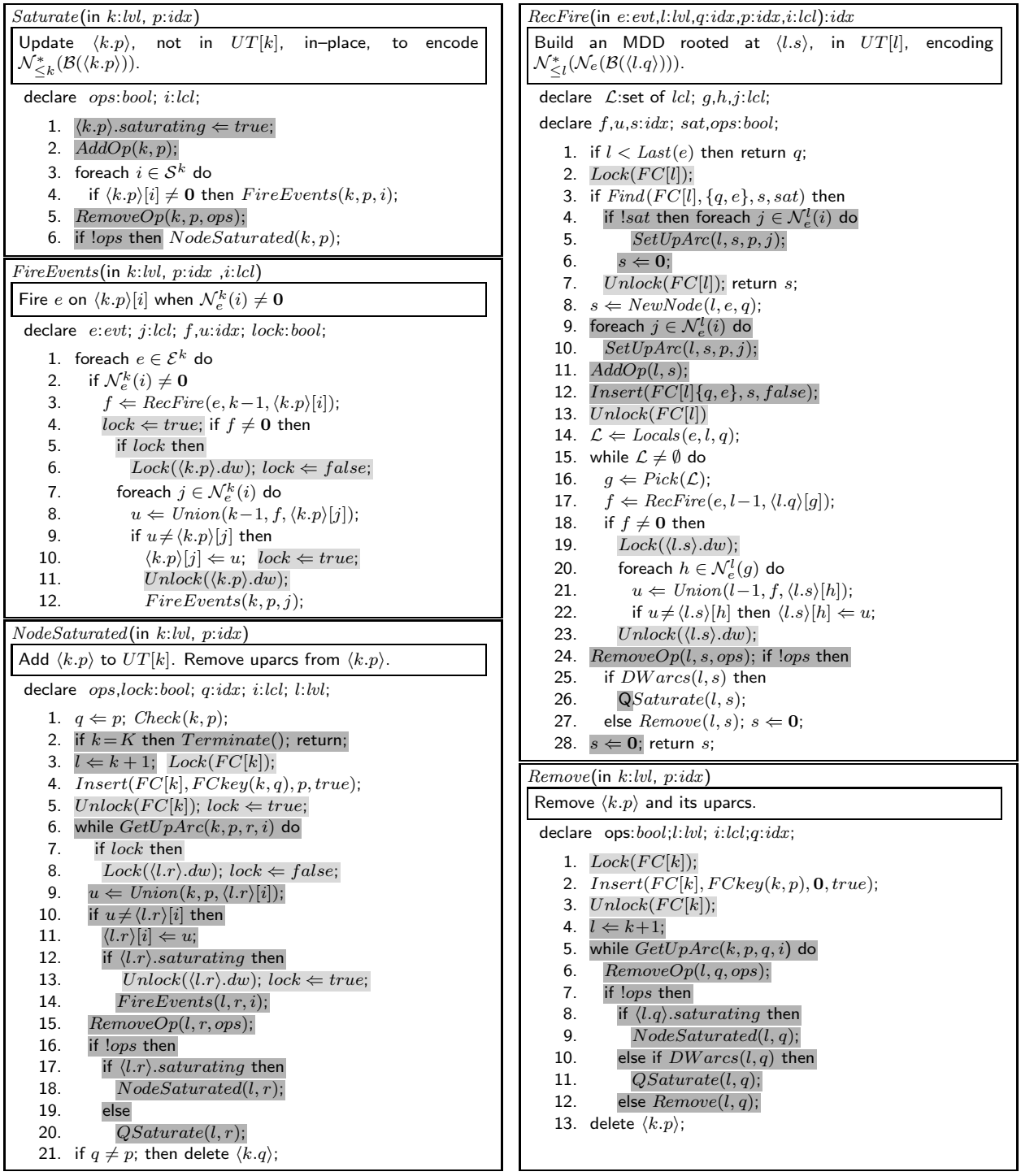


Figure 1: *Pseudo–code for the parallel node–saturation algorithm.*

the saturation task is being performed by a thread, it registers the thread on the node via *AddOp*. It begins the process of exhaustively firing events on the node by calling *FireEvents* for each non zero state. Once it has fired the events the task is complete and it calls *RemoveOp*. The function allows the thread to check the status of the node to see whether it is saturated. It can continue work on any nodes dependent upon the node reaching a

Gen(in s:array[1..K] of *lcl*, *nthr*:*int*):*idx*

Create *nthr* threads. Build an MDD rooted at $\langle K.root \rangle$ encoding the state space and return *root*, in $UT[K]$.

Union(in *k*:*lvl*, *p*:*idx*, *q*:*idx*):*idx*

Build an MDD rooted at $\langle k.s \rangle$, in $UT[k]$, encoding the Union of $\langle k.p \rangle$ $\langle k.q \rangle$. Return s.

DWarcs(in *k*:*lvl*, *p*:*idx*):*bool*

If $\langle k.p \rangle[i] \neq \mathbf{0}$ for any local state at level k return $true$ otherwise return $false$;

SetUpArc(in *k*:*lvl*, *p*:*idx*, *q*:*idx*, *i*:*lcl*)

Lock ($\langle k.p \rangle.ua$). Add an arc (q,i) to the end of the list of upward arcs for $\langle k.p \rangle$; $AddOp(k+1, q)$; Unlock ($\langle k.p \rangle.ua$);

GetUpArc(*k*:*lvl*,*p*:*idx*,out *q*:*idx*,*i*:*lcl*):*bool*

If the list of upward arcs is not empty retrieve and remove (q,i) from head of list and return $true$. Otherwise return $false$.

AddOp(in *k*:*lvl*, *p*:*idx*)

Add an operation to $\langle k.p \rangle$. $Lock(\langle k.p \rangle.ops)$. Increment $\langle k.p \rangle.ops$. $Unlock(\langle k.p \rangle.ops)$.

RemoveOp(in *k*:*lvl*, *p*:*idx*, out *op*:*bool*)

Remove operation from $\langle k.p \rangle$. $Lock(\langle k.p \rangle.ops)$. Decrement $\langle k.p \rangle.ops$. If $\langle k.p \rangle.ops = 0$ set *op* to false otherwise set *op* to true. $Unlock(\langle k.p \rangle.ops)$.

ThreadLoop()

If there are no items in the task queue sleep until woken up. Otherwise remove the head item (k, p) from the task queue. If (k, p) is $(\mathbf{0}, \mathbf{0})$ call $Terminate()$ and terminate the thread, otherwise call $Saturate(k, p)$.

Find(in *tab*, *key*, out *v*, *sat*:*bool*):*bool*,

If (key, x, y) is in hash table *tab*, set v to x and sat to y and return $true$. Else, return $false$.

Insert(inout *tab*, in *key*, *v*, *sat*:*bool*)

If *key* is not $(\mathbf{0}, \mathbf{0})$ insert (key, v, sat) in hash table *tab*, if it does not contain an entry $(key, \cdot, true)$.

Locals(in *e*:*evnt*, *k*:*lvl*, *p*:*idx*):set of *lcl*

Return all of the local states in p locally enabling e. If there are no states in p locally enabling e then return $\emptyset$.

Pick(inout $\mathcal{L}$:set of *lcl*):*lcl*

Remove and return an element from $\mathcal{L}$.

NewNode(in *k*:*lvl*,*e*:*evt*,*q*:*idx*):*idx*

Create $\langle k.p \rangle$ with arcs set to $\mathbf{0}$, set the *key* (e,q) for $\langle k.p \rangle$, return p.

Check(in *k*:*lvl*, inout *p*:*idx*)

If $\langle k.p \rangle$, not in $UT[k]$, duplicates $\langle k.q \rangle$, in $UT[k]$, delete $\langle k.p \rangle$ and set p to q. Else, insert $\langle k.p \rangle$ in $UT[k]$. If $\langle k.p \rangle[0] = \cdots = \langle k.p \rangle[n^k - 1] = \mathbf{0}$ or $\mathbf{1}$, delete $\langle k.p \rangle$ and set p to $\mathbf{0}$ or $\mathbf{1}$.

FCkey(in *k*:*lvl*, *p*:*idx*):*key*

Return the *key* for $\langle k.p \rangle$.

QSaturate(in *k*:*lvl*, *p*:*idx*)

Add item (k, p) to the task queue. Request any sleeping threads wake up.

Terminate()

Add item $(\mathbf{0}, \mathbf{0})$ to the task queue. Request any sleeping threads wake up.

Figure 2: *Supporting function descriptions for the parallel node–saturation algorithm.*

fixpoint.

FireEvents: checks to see if any events are enabled in the state being fired upon and calls *RecFire* to fire an enabled event. Successful firings result in the node being updated with the work carried out by the firing. Any updated states invoke a recursive call to *FireEvents* on the updated state.

RecFire: Uncompleted nodes discovered in the cache have upward arcs set from them to the calling node via *SetUpArc*. For new work, a node is created setting the *FC key* in the process. The thread registers with the new node and adds it to the *FC* as a work request. Upward arcs are set to the calling node. *RecFire* is recursively called to continue event firing then the thread de-registers from the node. Nodes at the bottom of the MDD generated by the event firing are either added to the task queue or removed if the event is disabled.

NodeSaturated: is called when a node is saturated. The node is checked into the unique table. It updates nodes dependent upon the saturated node via upward arcs, and allows the thread to continue working on them. The termination condition occurs when this function is called for the top level node.

3.2 Algorithm Example

We illustrate the parallel algorithm with an example for a thread pool of two threads shown in Figures 3 and 4. The disjunctively partitioned transition relation is represented as an *event matrix* shown in 3.*a* with four events.

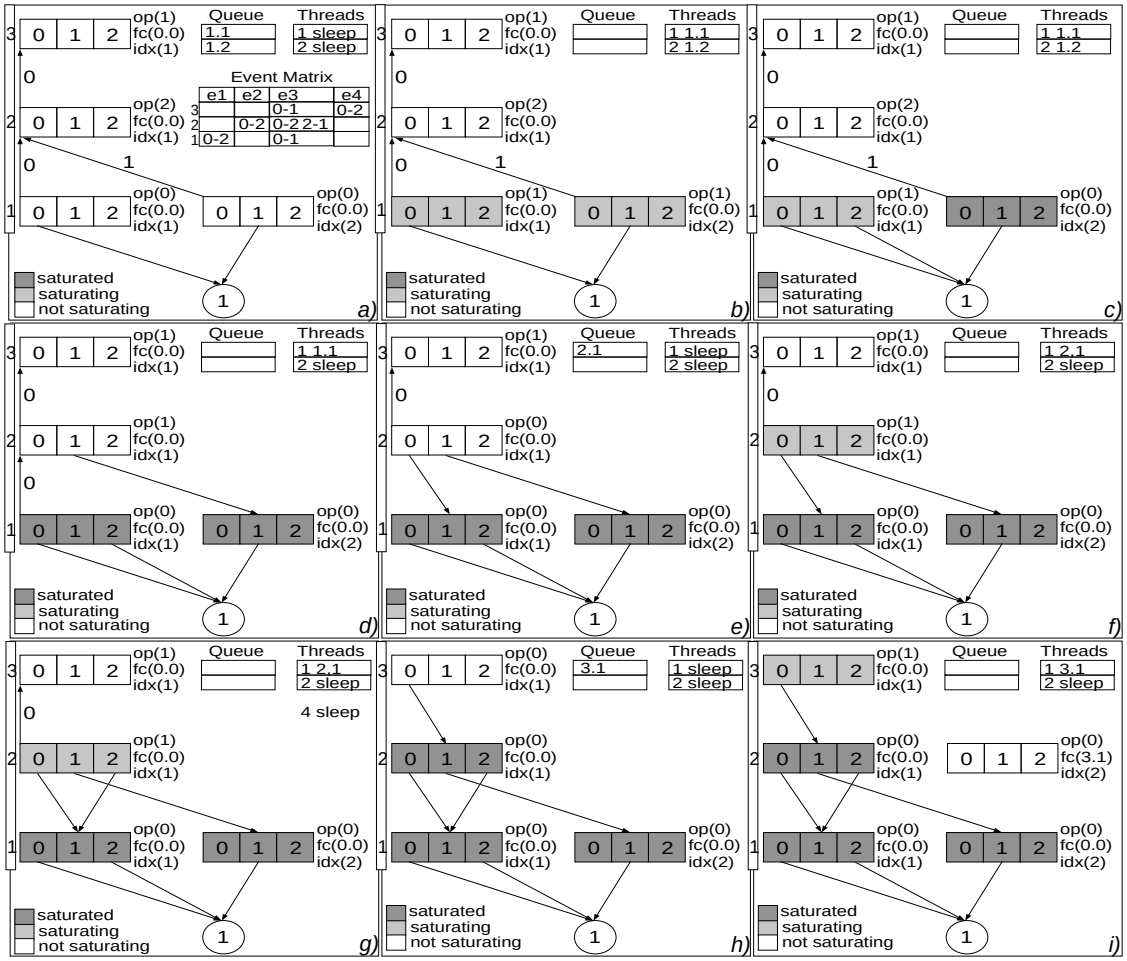


Figure 3: *Parallel node-saturation algorithm example (part 1)*

a) *Gen* generates an initial MDD. All nodes are marked as *not saturating*. None of the nodes have an FC key (*fc*) since the nodes are not created from a *RecFire* operation. Operations (*op*) is incremented for each upward arc set on a node. *Saturate* tasks are added to the task queue for bottom nodes of the MDD, $\langle 1.1 \rangle$ and $\langle 1.2 \rangle$. The sleeping threads are about to be woken by the new tasks.

b) The threads have woken and removed the tasks from the queue. They both call *Saturate* which marks each target node as *saturating* and increments *op* to indicate they are currently being operated upon by the *Saturate* task.

c) *Saturate(1,1)* makes a local update on $\langle 1.1 \rangle$ by firing event *e1*. *Saturate(1,2)* has completed since no events are enabled on $\langle 1.2 \rangle$ and has decremented *op* to zero. Since *op* is zero *NodeSaturated* is called which marks the node as *saturated* and checks the node into the *unique table*.

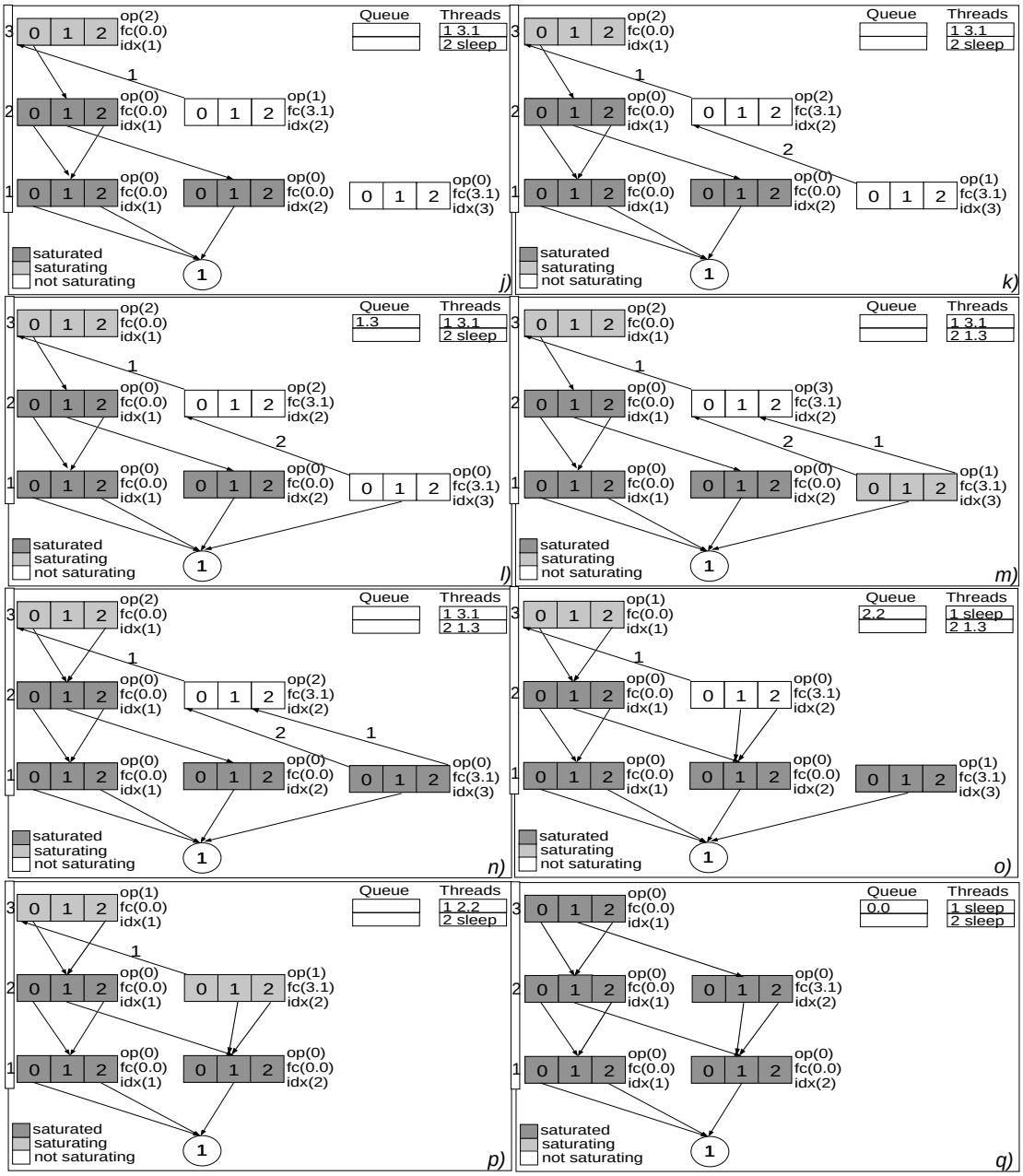


Figure 4: *Parallel node-saturation algorithm example (part 2)*

d) *Saturate(1,1)* has completed firing events and decremented *op* to zero, marking the node as *saturated*. *NodeSaturated(1,2)* removes the upward arc to $\langle 2.1 \rangle$ and replaces it with a downward arc, decrementing *op* for $\langle 2.1 \rangle$ in the process. Since *op* is nonzero *NodeSaturated* terminates leaving thread 2 to sleep.

e) *NodeSaturated(1,1)* removes the upward arc to $\langle 2.1 \rangle$ and replaces it with a downward arc and decrements the *op* for $\langle 2.1 \rangle$. Since *op* is now zero and the node is not *saturating*, a new *Saturate* task is added to the task queue for $\langle 2.1 \rangle$. *NodeSaturated* completes allowing the thread to return to sleep.

f) Thread 1 is woken up by the new task and removes it from the queue. It calls *Saturate(2,1)*

which increments *op* and marks $\langle 2.1 \rangle$ as *saturating*.
g) Saturate(2,1) makes an in-place-update on $\langle 2.1 \rangle$ firing *e2* to set local state 2 to point to node $\langle 1.1 \rangle$.
h) Saturate(2,1) completes firing, decrementing *op* and calls *NodeSaturated* which replaces the upwards arc to $\langle 3.1 \rangle$ with a downward arc and decrements *op*. Since $\langle 3.1 \rangle$ is not *saturating* and has no *op* a task to *Saturate* $\langle 3.1 \rangle$ is added to the queue. *NodeSaturated* terminates allowing thread 1 to return to sleep.
i) Thread 1 is woken by the addition of the task to the queue. It calls *Saturate(3,1)* which marks the node as *saturating* and increments *op*. *RecFire* is called for event *e3* which creates node $\langle 2.2 \rangle$ which contains an FC key *fc(3.1)*.
j) *RecFire* increments *op* on $\langle 2.2 \rangle$ and sets an upward arc to $\langle 3.1 \rangle$ incrementing *op* in the process. It recursively calls *RecFire* which creates $\langle 1.1 \rangle$
k) *RecFire* increments *op* on $\langle 1.3 \rangle$ and sets an upward arc to $\langle 2.2 \rangle$ incrementing *op* in the process.
l) *RecFire* sets a downward arc from $\langle 1.3 \rangle$ to **1** and terminates decrementing *op* in the process. On termination, since *op* is zero and the node is *not saturating* a task to *Saturate* $\langle 1.3 \rangle$ is added to the task queue.
m) Thread 2 picks up the new task and calls *Saturate(1,3)* which marks the node as *saturating* and increments *op*. Meanwhile Thread 1 has continued with *RecFire* and has discovered $\langle 1.3 \rangle$ as an *unsaturated* node in the firing cache while firing *e3* subsequently setting an upward arc to $\langle 2.2 \rangle$ and incrementing *op*.
n) $\langle 1.3 \rangle$ has completed saturating and decrements *op* to 0 and marks the node as *Saturated*. Meanwhile *RecFire* has completed on $\langle 2.2 \rangle$, decrementing *op*, and *Saturate* continues on $\langle 3.1 \rangle$ firing *e4* to make a local update.
o) *Saturate* has completed on $\langle 3.1 \rangle$ decrementing *op* and returning thread 1 to sleep. Since *op* is greater than 0, the node is not yet saturated. *NodeSaturated* has been called on $\langle 1.3 \rangle$ which has discovered the node is the same as $\langle 1.2 \rangle$ while checking it into the *unique table*, and has removed the upward arcs, setting the downward arcs to this node. Since *op* is 0, a *Saturate* task is added to the queue for $\langle 2.2 \rangle$ and the thread goes to sleep.
p) Thread 1 takes the *Saturate* task for $\langle 2.2 \rangle$, setting the thread to *saturating* and incrementing *op*.
q) *Saturate* completes on $\langle 2.2 \rangle$ decrementing *op* to zero and calling *NodeSaturate* which replaces the upward arc to $\langle 3.1 \rangle$ with a downward arc and decrements *op*. Since *op* is 0 and the node is *saturating* the node is *Saturated*. Since this is the root node, *Terminate* is called which adds (0.0) into the task queue, which terminates the threads. The final state space is shown.

3.3 Correctness of the Algorithm

The algorithm in Figure 1 can be expressed in terms of its sequential counterpart. Removing the highlighted parallel code gives us the correct version of the sequential algorithm. The correctness can be shown by demonstrating that the parallel code allows the algorithm to arrive at the same result and locks prevent any data races. We can illustrate the calling structure of both versions using Figure 5 as an example of the calling order of functions for applying the event matrix to the MDD in Fig. 5a. The call graphs in Fig. 5b and Fig. 5c are the calling order of functions for the sequential and parallel code respectively, where Fig. 5d

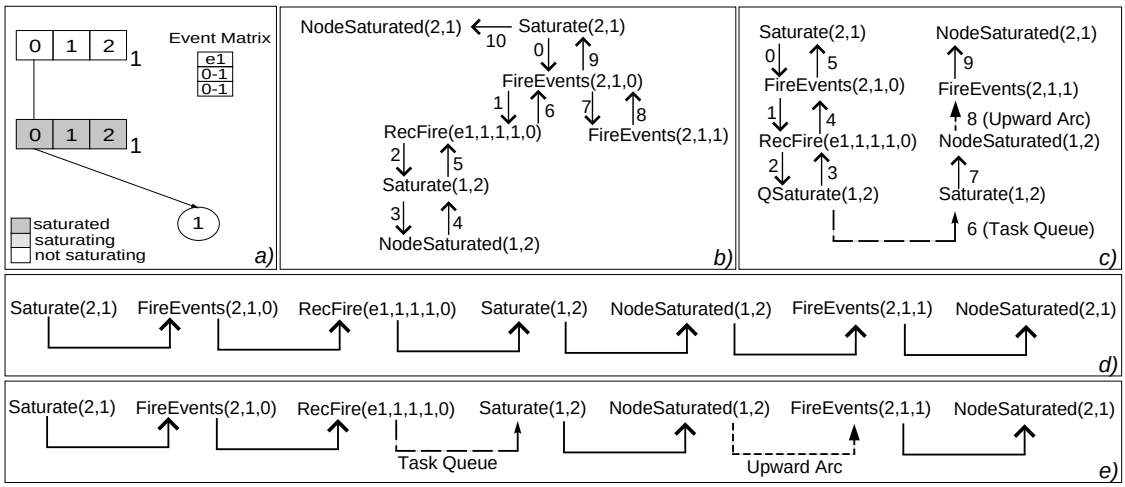


Figure 5: *The calling order of functions for the sequential and parallel algorithms*

and Fig. 5e further simplify the order. Function calls in the sequential version are directly replaced by the task queue and upward arcs in the parallel version. Since locks ensure that updating the node is atomic, firing events exhaustively will result in the same MDD shape for the saturated node as the sequential version.

The locks prevent data races in a number of places. Lines 4-6 and 10-11 of *FireEvents*, lines 19 and 23 of *RecFire* and lines 8 and 13 of *NodeSaturate* ensure that any updates made to the downward arcs during a union operation require that a lock for the downwards arc needs to be gained first. The combined union and update of a node is therefore an atomic operation. The locking of the firing cache ensures that data races are eliminated for the firing cache for both *unsaturated* and *saturated* nodes. Lines 2 and 7 and 13 of of *RecFire* ensure that when a node is found in the firing cache it is up to date, and if an *unsaturated* node is inserted at line 12 then the firing cache remains up to date as the lock has not been released. The node cannot transition from *unsaturated* to *saturated* until the lock has been gained in line 3 of *NodeSaturated* and line 1 of *Remove* since lines 4 and 2 of the respective functions make the transition by inserting the *saturated* node into the firing cache. This also ensures the the upward arcs set in lines 5 and 10 of *RecFire* is correct since the task will only be able to set the upward arcs if the firing cache lock is gained. The registering and de-registering of tasks is protected in the *AddOp* and *RemoveOp* function since the functions must gain a lock before writing and reading the number of tasks. The *saturating* flag in line 1 of *Saturate* does not require protection however, since this can implicitly be the only task operating on the the node at the time. Locks also exist for access to the *Unique table* and *Union cache*. All other data is unshared.

4 EXPERIMENTAL RESULTS

We built a prototypical parallel algorithm using C and the POSIX `Pthreads` library. To evaluate the algorithm we measured several aspects of the algorithms performance when utilizing between one and four cores on a shared-memory machine. For comparison we also measured the performance of a C version of the sequential algorithm on the same machine. The machine used for evaluation is a dual core, dual processor machine with 2GB

Table 1: *Experimental Results*

	Sequential		***1 Core***		***2 Cores***		***3 Cores***		***4 Cores***	
Slotted Ring (*Avg. Density* 0.40 *Avg. Events Per Level* 3.00)										
N	time	mem(b)	time	$r$mem	time	$r$mem	time	$r$mem	time	$r$mem
60	2.36	2002880	7.01	15.22	7.72	23.67	7.08	23.67	6.27	22.02
90	7.56	6109680	23.11	16.12	26.40	25.85	24.59	25.68	20.76	22.61
120	17.56	13726480	55.16	16.61	64.48	26.86	60.43	26.86	50.71	24.17
150	34.28	25933280	110.23	16.93	128.71	27.50	121.09	27.42	99.94	24.47
Random A (*Avg. Density* 0.40 *Avg. Events Per Level* 2.8)										
#	time	mem(b)	time	$r$mem	time	$r$mem	time	$r$mem	time	$r$mem
1	1.58	3665040	2.43	7.71	1.81	7.89	1.58	7.91	1.49	7.96
2	4.90	4603360	31.24	20.53	20.95	20.91	18.61	21.26	17.41	21.39
3	12.09	8927840	76.60	20.39	51.22	21.29	49.20	22.97	46.81	23.38
4	7.70	7813120	16.52	12.69	11.03	12.95	9.70	13.25	8.75	13.25
Round Robin (*Avg. Density* 0.19 *Avg. Events Per Level* 4.96-4.98)										
N	time	mem(b)	time	$r$mem	time	$r$mem	time	$r$mem	time	$r$mem
150	6.03	86462784	10.98	2.26	7.64	2.26	7.71	2.26	7.75	2.26
180	12.26	147830244	21.44	2.27	14.39	2.27	14.59	2.26	14.69	2.26
210	23.51	232961304	38.89	2.27	25.21	2.27	25.65	2.27	25.74	2.27
240	41.73	345743964	65.89	2.27	41.88	2.27	42.32	2.27	42.55	2.27
Random B (*Avg Density* 0.20 *Avg. Events Per Level* 4.41-4.54)										
#	time	mem(b)	time	$r$mem	time	$r$mem	time	$r$mem	time	$r$mem
5	6.99	44951280	12.40	3.38	9.25	3.42	8.27	3.39	8.27	8.07
6	15.20	73501020	26.81	3.43	18.66	3.46	16.76	3.44	16.00	3.45
7	16.90	55277700	30.53	4.67	21.53	4.70	19.62	4.68	18.76	4.67
8	1.62	17186280	3.15	3.34	2.51	3.35	2.35	3.33	2.28	3.32
Kanban (*Avg Density* 1.67-1.82 *Avg. Events Per Level* 1.50)										
N	time	mem(b)	time	$r$mem	time	$r$mem	time	$r$mem	time	$r$mem
15	0.64	1008188	1.15	2.74	0.71	3.02	0.70	3.56	0.70	4.70
20	2.85	3601728	5.98	2.77	3.52	3.05	3.34	4.19	3.32	5.32
25	10.08	9934868	22.91	2.80	12.79	3.22	12.31	4.77	12.27	6.65
30	29.73	23089108	73.24	2.83	39.96	3.36	37.32	5.33	36.65	6.74
Random C (*Avg Density* 1.74 *Avg. Events Per Level* 1.50)										
#	time	mem(b)	time	$r$mem	time	$r$mem	time	$r$mem	time	$r$mem
9	5.93	1378080	33.90	10.6	30.47	10.8	51.25	11.91	41.72	11.29
10	4.20	503440	5.33	7.65	3.11	7.76	2.50	7.90	2.32	8.00
11	7.02	874640	20.93	15.96	10.02	16.58	8.36	18.86	7.51	19.37
12	5.39	642640	10.43	10.54	6.03	10.92	4.78	10.65	4.17	10.60

of memory and Intel Xeon CPU 3.06GHz processors with 512kb cache sizes running Redhat Linux AS. We applied the algorithm to the Slotted Ring [18], Round Robin [9], Kanban and a number of *randomly generated* models. The traditional models have been used to previously evaluate the sequential algorithm [5]. In our results we have classified the event matrices by average *density*, i.e. the number of state updates within an event relative to the

width of a node, and the average number of *events per level*. Based on these properties, we generated random models with a varying number of randomly generated events with similar matrix characteristics to the traditional models. Run-times and relative memory against the sequential version are shown in Table 1, where N is either the number of nodes in a Slotted Ring network, the number of processes for the Round Robin protocol or the tokens in the Kanban system.

Traditional Models: All of the results show speedups when running our parallel algorithm on four cores instead of one. The best parallelism is for a Kanban model with an approximate 50% speedup. Despite this however, the algorithm is still slower than the sequential version. The algorithm is most comparable to the sequential version for the Round Robin model with only a slight decrease in run-time for the highest value of N. The memory consumption of the parallel algorithm is greater for all models, varying between up to around 28 times for the Slotted Ring model to less than 3 times for the Round Robin model.

Random Models: The results vary between speedups and slowdowns. The majority of the models show speedups when running the parallel algorithm on 4 cores instead of 1 core, but remain slower than the sequential version. Model 9 shows a slowdown of approximately 30% when running on 4 cores instead of 1. Model 10 shows the best parallelism with a speedup over 40%. Memory consumption increases for the parallel algorithm on all models and varies between approximately 5 and 25 times that of the sequential version.

Much of the extant research examines run-time, memory consumption and direct measurements on the state space for evaluating parallel algorithms. Our approach to evaluation is more thorough. We used several tools to evaluate the performance of the algorithm. We carefully selected our architecture in order to allow Intel Threading Tools to be used on the algorithm. Using the Thread Checker we verified that the locks were set correctly to avoid data races and would not interfere with the results. Using the Thread Profiler we obtained measurements of the parallel overheads, and how well the cores were utilized by the algorithm. To investigate the parallel effect on the construction of the state space we built a tool to visualize the MDD construction. To evaluate any code overheads we used the GNU Profiler [10] to profile the code. Combining these tools with our results and direct measurements provided us with a great deal of insight into the performance of the algorithm.

Overheads: The run-time results and memory are affected by the overheads incurred by the parallel algorithm. High overheads prevent the parallel algorithm from competing with the sequential version. We saw two types of overhead, the parallel overhead incurred by the introduction of locks and threads, and the code overhead incurred from removing mutual recursion from the algorithm. The highest memory and run-time overhead comes from the use of upward arcs.

Extra Work: The order in which the events are fired affects the amount of work the algorithm has to perform. Due to the dependencies between events, parallel events are often fired on smaller state sets than the sequential version. This creates more work, and larger intermediate MDDs. The extra work can outweigh the benefits of parallelism. It also introduces higher overheads.

Parallelism: The number of parallel events and how well this causes the work to branch in parallel during construction affects the level of parallelism of the algorithm. The lower the parallelism, the lower the number of parallel tasks to perform. Low parallelism means cores are undersubscribed during construction.

We examined the event matrices as to whether overheads, extra work and parallelism could be determined by their properties. We chose to classify our models by properties that could affect parallelism. Our evaluation showed that the factors which affect whether the algorithm showed speedups are more complicated than the properties we chose. To determine how well a model can be parallelized we also need to look at how event orderings affect parallelism. Orderings can introduce spikes in performance between processors which has been seen before in explicit parallel model checking [14]. This effect is greater in our algorithm since individual event firings produce larger state sets.

Our results are encouraging for models that have low overheads, are unaffected by extra work and show good parallelism. The overheads incurred by parallelism are too great however to show speedups over the sequential version. Given that Saturation is orders of magnitude more time- and memory-efficient than other symbolic algorithms [6], it would be difficult for our parallel algorithm to further improve over the sequential version. When we incur several penalties from parallelization such as extra work, code to remove mutual recursion and parallel overheads, parallelism is likely to hinder rather than enhance the state space construction process.

5 RELATED WORK

For explicit based state space generators the algorithm is the key consideration for parallelization [14]. For symbolic state space generators the complex data structure for storing states often needs to be considered [6, 11, 12, 13, 17, 19]. The extant research on symbolic model checking has focused primarily on networks of workstations (NOW). We can classify the symbolic parallelization approaches into a number of distinct categories and show that the category that our work fits into is unique from the previous work in this area.

Data Parallelization (Memory). Most of the work on parallel symbolic state space generation considers only how to parallelize the data structure. These approaches target the increased memory available on NOW by slicing the data structure and distributing it across processors of the NOW. The structure has previously been sliced horizontally [6] and vertically [13, 17, 19]. Horizontal slicing scales well but prevents the state space generation task from being speeded up since each slice has to complete its work before the next slice begins its work. Finding a good vertical slicing is a non-trivial issue often leading to poor scalability. In order to facilitate scalability, load balancing techniques need to be employed. The most advanced work in this area employs workstealing techniques to distribute the work dynamically [12].

Data and Algorithm Parallelization (Memory/Time). More recently researchers have parallelized the algorithm in order to gain speedups from developing vertical slices on different processors of a NOW [11]. If the algorithm developing the slices has to frequently synchronize on the application of the imaging function, each round of computation is only as fast as the slowest time it takes for a slice to develop on a processor. In order to achieve speed-ups the research tackles the difficult task of removing the synchronous nature of the algorithm. The parallel algorithm allows slices to develop asynchronously while the imaging function is applied to create more work. The work is load balanced using the workstealing techniques developed in [12]. For very large circuits this technique has proved to be a very efficient parallelization showing up to an order of magnitude improvement in time-efficiency .

Utilizing Idle Processors (Memory/Time). Recent work has also considered ways to utilize idle processors during state space construction [2]. The idle processors are used to perform and cache work that may be performed in future while a main processor develops the state space. If work that the main processor requires has already been performed by another processor the main processor retrieves it from the cache. This reduces the peak size of the data structure during state space construction and improves time-efficiency if the amount of utilized work performed by the idle processors is sufficient to overcome the overhead of allocating work to the processors and synchronizing on the cache.

Our Approach: Algorithm Parallelization (Time). Our approach is unique in that we consider how to parallelize only the algorithm itself. The primary goal is improving time-efficiency by utilizing the extra processing power. Parallel overheads are addressed while leaving the data structure whole during state space construction. A shared memory architecture is targeted in order to reduce the costs of synchronization. In contrast to most other work with the exception of [14] we have evaluated the performance characteristics of our parallel algorithm very carefully.

In [14] a detailed study of a shared memory architecture is used to parallelize an explicit state space generation algorithm. The approach employs workstealing techniques in order to load balance. Many of the parallelization overheads are addressed by tailoring the parallelization specifically to the selected architecture. The high optimization of the algorithm for the architecture allows the parallel algorithm overcome parallel overheads, showing good linear speedups for several models.

6 CONCLUSIONS AND FUTURE WORK

We investigated whether the MDD-based Saturation algorithm for computing reachable state spaces of asynchronous system models can be parallelized on shared-memory architectures, such as multi-processor, multi-core PCs. This is a challenging question since symbolic state-space generation is an irregular task.

The idea for parallelizing Saturation was to consider the firing of events on a node as a task. Because Saturation is a mutual recursive algorithm, many relatively lightweight tasks are created which cannot be managed efficiently by the operating system. Instead, we implemented a task queue, which stores tasks awaiting processing, ourselves. Available threads running on dedicated processors then collect work from this queue, thereby minimizing processor idle time. Our conceptual ideas and implementation strategy for the thread pool are not specific to Saturation, and are thus reusable for implementations of other parallel algorithms. We showed speedups for traditional models utilizing four computing cores over a single core of up to 50%. However, speed-ups over the original sequential version of Saturation depend very much on the specific model studied.

Future work shall proceed along three orthogonal directions. Firstly, we wish to optimize our current implementation and explore heuristics for the order in which tasks are taken out of the thread pool. This order does not affect the correctness of our parallel algorithm but significantly its efficiency. Secondly, it should be investigated whether our ideas can be combined with those of [1] for efficiently parallelizing Saturation for distributed-memory architectures, such as PC clusters. Thirdly, other approaches to further exploiting modern parallel computer architectures shall be explored, such as the predictive firing of events suggested in [2].

REFERENCES

[1] M.-Y. Chung and G. Ciardo. Saturation NOW. *QEST*, pp. 272–281, 2004.

[2] M.-Y. Chung and G. Ciardo. A dynamic firing speculation to speedup distributed symbolic state-space generation. In *IPDPS*. IEEE, 2006.

[3] G. Ciardo, R. Jones, A. Miner, and R. Siminiceanu. SMART: Stochastic model analyzer for reliability and timing. *Tools of Measurement, Modelling and Evaluation of Computer-Communication Systems*, pp. 29–34, 2001.

[4] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Efficient symbolic state–space construction for asynchronous systems. In *ICATPN*, vol. 1839 of *LNCS*, pp. 103–122, 2000.

[5] G. Ciardo, G. Lüttgen, and R. Siminiceanu. Saturation: An efficient iteration strategy for symbolic state-space generation. In *TACAS*, vol. 2031 of *LNCS*, pp. 328–342, 2001.

[6] G. Ciardo and R. Siminiceanu. Structural symbolic CTL model checking of asynchronous systems. In *CAV*, vol. 2725 of *LNCS*, pp. 40–53, 2003.

[7] A. Cimatti, E. M. Clarke, F. Giunchiglia, and M. Roveri. NUSMV: A new symbolic model checker. *STTT*, 2(4):410–425, 2000.

[8] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT, 1999.

[9] S. Graf, B.Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *FAC*, 8(5):607–616, 1996.

[10] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler (with retrospective). In *Best of PLDI*, pp. 49–57. ACM, 1982.

[11] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *CHARME*, vol. 3725 of *LNCS*, pp. 129–145, 2005.

[12] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. In *CAV*, vol. 2725 of *LNCS*, pp. 54–66, 2003.

[13] T. Heyman, D. Geist, O. Grumberg, and A. Schuster. Achieving scalability in parallel reachability analysis of very large circuits. In *CAV*, vol. 1855, pp. 20–35, 2000.

[14] C. P Inggs. *Parallel Model Checking On Shared Memory Architectures*. PhD thesis, University of Manchester, UK, 2004.

[15] T. Kam, T. Villa, R. Brayton, and A. L. Sangiovanni-Vincentelli. Multi-valued decision diagrams: Theory and applications. *Multiple-Valued Logic*, 4(1-2):9–62, 1998.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.

[17] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *FMCAD*, vol. 1522 of *LNCS*, pp. 501–507, 1998.

[18] E. Pastor, O. Roig, J. Cortadella, and R. M. Badia. Petri net analysis using boolean manipulation. In *PNPM*, vol. 815 of *LNCS*, pp. 416–435, 1994.

[19] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *DAC*, pp. 641–644. ACM, 1996.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
01/02/2007	Contractor Report	

4. TITLE AND SUBTITLE

A Parallel Saturation Algorithm on Shared Memory Architectures

5a. CONTRACT NUMBER

5b. GRANT NUMBER
NCC1-02043

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Jonathan Ezekiel, Gerald Luettgen, and Radu I. Siminiceanu

5d. PROJECT NUMBER
2204

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

NASA Langley Research Center
Hampton, VA 23681-2199

National Institute of Aerospace (NIA)
100 Exploration Way
Hampton, VA 23666

8. PERFORMING ORGANIZATION REPORT NUMBER
NIA Report No. 2007-01

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITOR'S ACRONYM(S)
NASA

11. SPONSORING/MONITORING REPORT NUMBER
NASA/CR-2007-214543

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified - Unlimited
Subject Category 62
Availability: NASA CASI (301) 621-0390

13. SUPPLEMENTARY NOTES

Langley Technical Monitor: Ricky W. Butler

14. ABSTRACT

Symbolic state-space generators are notoriously hard to parallelize. However, the Saturation algorithm implemented in the SMART verification tool differs from other sequential symbolic state-space generators in that it exploits the locality of firing events in asynchronous system models. This paper explores whether event locality can be utilized to efficiently parallelize Saturation on shared-memory architectures. Conceptually, we propose to parallelize the firing of events within a decision diagram node, which is technically realized via a thread pool. We discuss the challenges involved in our parallel design and conduct experimental studies on its prototypical implementation. On a dual-processor dualcore PC, our studies show speed-ups for several example models, e.g., of up to 50% for a Kanban model, when compared to running our algorithm only on a single core.

15. SUBJECT TERMS

Model Checking; Parallel Algorithms; Shared Memory Architectures

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19b. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			STI Help Desk (email: help@sti.nasa.gov)
U	U	U	UU	21	19b. TELEPHONE NUMBER *(Include area code)* (301) 621-0390