Acceleration of Simulation in Odin II with OpenMP

by

Andrew Somerville and Kenneth B. Kent

TR 11-210, August 31, 2011

Faculty of Computer Science
University of New Brunswick
Fredericton, NB, E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
Email: fcs@unb.ca
http://www.cs.unb.ca

Contents

List of Figures

List of Tables

Nomenclature

V_{CC} - Common collector voltage

ABC - A logic synthesis system used as a logic optimizer

ACEnet - Atlantic Computational Excellence Network

AMD - Advanced Micro Devices. A manufacturer of CPUs

API - Application Programming Interface

AST - Abstract Syntax Tree

CAD - Computer Aided Design

CPU - Central Processing Unit

DDR2 - Double Data Rate, Version 2. A type of random access memory

FPGA - Field Programmable Gate Array

GHz - Gigahertz. On billion cycles per second

GND - A ground node

HDL - Hardware Description Language

LUT - Look-up Table

MHz - Megahertz. On million cycles per second

netlist - A directed graph representing a digital circuit

OpenMP - An open source threading API

Opteron 8000-Series - A server CPU based on AMD's K8 microarchitecture

Phenom X4 - A quad-core desktop CPU based on AMD's K10 microarchitecture

T-VPack - Technology-Versatile Packer. A logic block clustering tool

VPR - Versatile Placement and Routing. A placement and routing tool

Abstract

Odin II is a Verilog Hardware Description Language (HDL) synthesis and verification framework developed to assist Field Programmable Gate Array (FPGA) researchers in investigating approaches and improvements to various phases of HDL elaboration, as well as new FPGA architectures. This paper discusses the development of a parallel simulation algorithm for Odin II. We will discuss the design of this algorithm in Sections 3 and 4. Its implementation will be discussed in Section 5. In Section 6 we will demonstrate that this algorithm provides good speedup with up to four processors on desktop hardware and up to five processors on a node in a big-node supercomputing cluster using a suite of eleven benchmarks. In Section 6 we will also show that this algorithm tends to provide better speedup as the problem size increases with up to four processors.

1 Introduction

Odin II is a framework for Verilog Hardware Description Language (HDL) synthesis developed to assist researchers in investigating different approaches and improvements to different phases of HDL elaboration as well as the exploration of new Field Programmable Gate Array (FPGA) architectures [4]. Odin II accepts as input a verilog HDL file representing a digital circuit as well a FPGA architecture specification file containing a description of a particular FPGA architecture. The verilog file is first compiled into an Abstract Syntax Tree (AST). From the AST, a directed graph called a netlist is derived [5].

Odin II currently includes a simulator which allows researchers to perform verification of the functional correctness of the netlist produced by Odin II. The simulator takes the netlist as input. The netlist may contain basic logic gates, Look-up Tables (LUTs), and FPGA-specific hard blocks described by the aforementioned FPGA architecture specification. [5] Odin II then simulates the netlist over a set of test vectors, producing a corresponding set of output vectors.

This paper will focus on improving the performance of the Odin II simulator by making use of OpenMP, a cross-platform threading API which was chosen for its portability, simplicity, and flexibility [8]. A parallel approach to netlist simulation will be detailed in Section 3 and various design choices will be discussed in Section 4. The implementation of the parallel simulator will be discussed in Section 5. Results of running the parallel simulator on both a desktop machine and a node in a super computer using a suite of eleven benchmarks will be presented in Section 6. Finally, results will be summarized in Section 8 and conclusions regarding the performance of the parallel algorithm will be drawn in Section 9.

2 Background

Odin II is part of a CAD flow which includes three other tools: ABC, T-VPack, and VPR. ABC, developed at Berkeley, is a system for sequential synthesis and verification which is used as a logic optimizer [3]. T-VPack and VPR, developed at the University of Toronto, perform logic block clustering as well as placement and routing to a target FPGA, respectively [6] [7]. This CAD flow is depicted in Figure 1.

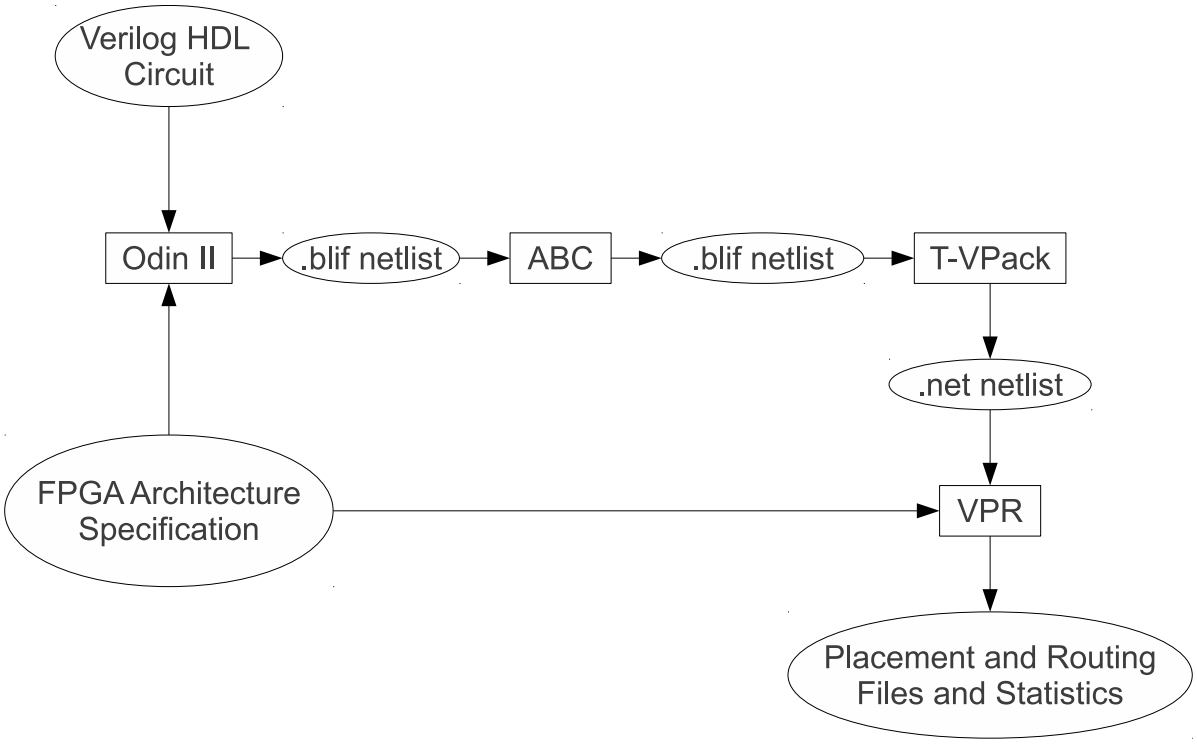

Figure 1. CAD Flow using Odin II and VPR [6]

A verilog HDL file is compiled by Odin II, which currently performs optimizations for hard block multipliers and hard block memories based on the FPGA architecture specification. Odin II can also perform simulation of the netlist at this stage. The output .blif netlist from Odin II is then optimized further by ABC. Finally T-VPack and VPR perform logic block clustering, placement, and routing of the circuit onto the FPGA specified in the FPGA architecture specification file [6].

The focus of this paper will be on the Odin II simulator, which is depicted in the block diagram

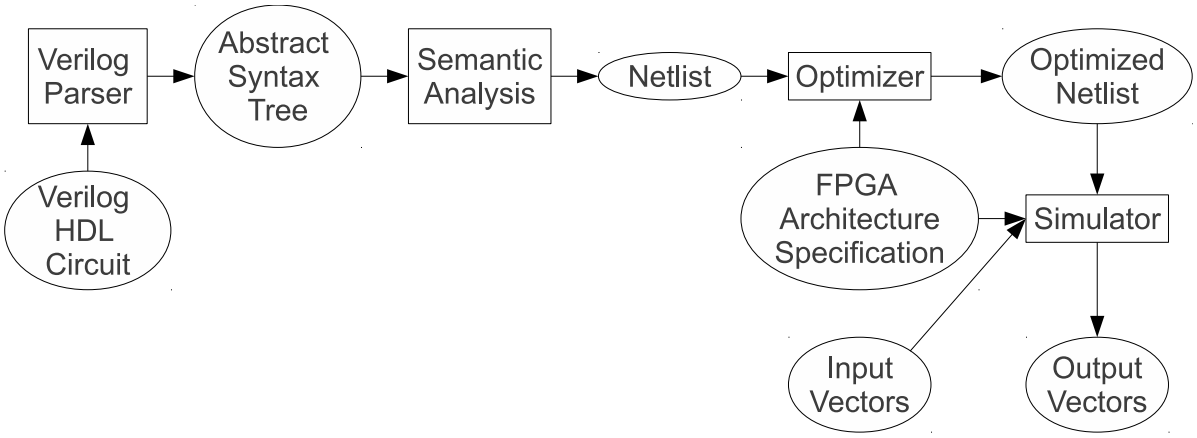

Figure 2. Design of Odin II, Showing the Simulator

in Figure 2. The simulator operates within Odin II, providing researchers with the ability to verify that the netlists produced by Odin II are functionally correct [6]. The simulator also makes use of the FPGA architecture specification file in order to simulate hard blocks. Currently multipliers, memories and generic black boxes are supported by the simulator [6].

2.1 The Odin II Simulation Process

The topmost nodes of the netlist represent the circuit's primary inputs and constant nodes. The bottommost nodes represent the primary outputs. During simulation, a set of input vectors is propagated from the primary inputs through the netlist cycle by cycle to attain a set of output vectors. Each pin has an associated cycle attribute, indicating the last cycle it completed. When each node is simulated, its input pins are used to compute the values of its output pins. As each pin is calculated, its cycle attribute is advanced to indicate that it has completed the current cycle of simulation.

At the beginning of a cycle, an input vector is either read in from a file or generated randomly. If the vector is generated, it is appended to a file for reference. This vector is applied to the primary input pins of the netlist. Flip-flops are then computed using values from the previous cycle. All ready top level nodes are then added to a master queue. Lastly, all ready constant nodes are added.

The nodes in the queue are then dequeued one by one and simulated by computing the values of their outputs and assigning those values to the output pins of the node. The cycle attribute of each pin is advanced as its value is assigned. After each node has been simulated, any nodes connected to its output pins are placed in the queue for simulation if they are not already in the queue, are ready for computation, and have not already had their values computed for this cycle. After each node is added to the queue, its *in_queue* attribute is set to prevent it from being added multiple times.

This process is repeated until there are no nodes left in the queue. After each cycle the values left on the output pins are appended to a file where they can be examined later. If there are still more input vectors to be simulated, the cycle is advanced and the next input vector is read. When there are no more input vectors, the simulator terminates.

3 A Parallel Approach

A parallel approach to netlist simulation was devised in order to take advantage of the prevalence of parallel computers. This particular approach divides the task of simulation into an initial set-up segment where top inputs and constant nodes are enqueued and a parallel segment where the queue is processed and the netlist is simulated.

Simulation makes use of two levels of queueing. A master queue is used as both a starting point for each thread and a fallback when a thread dries up. A local queue is used by each thread to traverse the segment of the netlist upon which that thread is currently working. While technically a single master queue could be shared by all threads, the use of separate local queues reduces the degree of synchronization required during simulation.

3.1 Initial Set-up - Sequential Segment

As with the sequential approach, an input vector is either read in from a file or generated randomly. This vector is then applied to the primary inputs of the netlist. Flip-flops are then computed using values from the previous cycle. All ready top level nodes are then added to the master queue. Lastly, all ready constant nodes are added.

While it is entirely optional from a functional standpoint, the immediate children of constant nodes are also added if they are ready for computation. This is done to help balance the computational load of the parallel section by providing threads with additional nodes high in the netlist to fall back on should they dry up. This will be referred to as pre-balancing (See Section 4.2).

At this stage, the master queue will contain only nodes which are ready for computation, eliminating the need for any readiness checks in the parallel section.

3.2 Processing the Queue - Parallel Segment

Execution is then split into p threads, where p is determined by OpenMP and is usually equal to the number of processors installed in the host system. Each thread constructs its own local

queue.

The first node fed into each thread is always pulled from the master queue. If the node has not already been completed, its outputs are computed. Otherwise the node is dropped and another node is dequeued. After the node's outputs are computed, the node's children are examined. Each ready child node which has not been computed and has not been added to another queue is added to the thread's local queue. Its *in_queue* attribute is set to prevent it from being added to multiple queues.

To help balance load across the CPUs, every p^{th} child node may be randomly selected to be added to the master queue rather than the thread's local queue. This is referred to as re-balancing (See Section 4.2). A node is then pulled from the local queue and the process is repeated. When the local queue is empty, each thread falls back on the master queue. When no more nodes exist in either the local queue or the master queue the thread terminates. When all threads terminate, the simulation process is complete.

3.3 Complexity

With a netlist consisting of n nodes visited in a depth-first manner - each top node being followed all the way down, in turn, the number of revisited nodes will depend on the degree of fan-in. Since each node not on the top-level will be visited once for every input pin, let d be the average degree of fan-in, (or the average number of input pins per node).

The total runtime of the simulation cycle will be the runtime of the sequential portion plus the runtime of the parallel portion. The runtime of the sequential portion depends primarily on the number of top input pins and constant nodes, which will be small compared to n, the total number of nodes in the netlist.

Assuming the initial setup has a runtime of $\mathcal{O}(1)$, the number of nodes visited by the algorithm will be:

$$T^*(n) = \mathcal{O}(n \cdot d)$$

but since each node is only calculated once, the number of nodes calculated will be

$$\mathcal{O}(n).$$

Because the computation of each node is a relatively trivial task, the time to visit all nodes in the netlist is expected to dominate as the problem size increases. Therefore $T^*(n)$ will be used as the algorithm's sequential complexity.

3.3.1 Parallel Complexity

With p processors, ideal load balancing, negligible threading overhead, and negligible synchronization wait time, the parallel simulator will compute $\mathcal{O}(n/p)$ nodes on each processor, and will examine $\mathcal{O}(d/p)$ child nodes connected to them.

This puts the parallel run time on the order of:

$$T_p(n) = \mathcal{O}\left(\frac{n}{p} \cdot \frac{d}{p}\right)$$

or

$$T_p(n) = \mathcal{O}\left(\frac{n \cdot d}{p}\right).$$

In the worst case of imbalance where one processor does all the work, its run time will be:

$$\mathcal{O}(n \cdot d)$$

which is equal to the sequential run time. Therefore attaining good speedup as the number of threads increases depends heavily on how well the load is balanced between threads.

3.4 Speedup and Efficiency

With p processors, this algorithm can be expected to achieve an ideal maximum speedup of:

$$S_p(n) = \frac{T^*(n)}{T_p(n)} = \frac{\mathcal{O}(n \cdot d)}{\mathcal{O}\left(\frac{n \cdot d}{p}\right)} = p$$

assuming a perfectly balanced workload, and negligible overhead and synchronization cost [11]. This gives us a theoretical maximum efficiency of

$$E_p(n) = \frac{S_p(n)}{p} = 1$$

which is ideal [11].

4 Design Considerations

The parallel simulator described in Section 3 leaves open several possibilities in terms of design choices. These include the use of synchronization and the employment of various approaches to load balancing. In this section we will discuss some of these choices and lay the foundation for exploring their relative effectiveness.

4.1 Synchronization

One key area where synchronization is required is around the master queue, which must be shared safely amongst all threads. If relatively few nodes are initially queued in the master queue compared to the number in the netlist, the chance of blocking will be small. Each processor will be working mostly from its local queue and only occasionally falling back on the master queue.

This synchronization point could be eliminated entirely by using an array as a global data structure, with fixed ranges assigned to each processor. But that approach would preclude the possibility of using the central structure as a dynamic load balancing mechanism. For this reason, a queue was deemed a better choice for a global data structure, despite the overhead incurred by synchronization.

Alternately the master queue could be used for all node queueing, eliminating the local queues altogether. While this would greatly diminish (if not eliminate) the need for load balancing, it would increase the number of master queue synchronization events to

$$\mathcal{O}(n)$$

increasing the complexity of synchronization to that of the sequential algorithm's node computation complexity. Since this mutual exclusion surrounds a central shared data structure, it would not distribute well across multiple CPUs. It would therefore eventually dominate the parallel run time as the number of processors approaches d.

This is a classic tradeoff between idle time brought on by load imbalance and synchronization overhead [10]. A master queue used sparingly represents a compromise between the two approaches in this case.

4.1.1 Locking of Individual Nodes

Since each child node upon being discovered must first be tested to determine if it has already been placed in a queue, and subsequently updated to indicate that it has, it makes some sense to lock the node between these two steps to ensure that it does not end up in multiple queues. The overhead of locking and unlocking every node every time it is discovered will be roughly:

$$\mathcal{O}(n \cdot d)$$

or the same as the algorithm as a whole, assuming that the likelihood of lock contention is negligible. It should distribute well across multiple processors, leading to a parallel complexity of:

$$\mathcal{O}\left(\frac{n \cdot d}{p}\right)$$

which is the same as the parallel algorithm ($T_p(n)$). Since this overhead is significant, and the period of time between the test and set of the *in_queue* flag is fairly short, in practise it is probably better to risk occasionally double queueing a node rather than incurring this overhead. Therefore the choice was made to omit this lock.

4.2 Load Balancing

In an idealized circuit, the processors would each start from a few of the primary inputs and constant nodes, and work their way to a primary output, each computing a roughly equal share of the nodes. Figure 3 illustrates a simple example. If CPU 1 stays a little ahead of CPU 0, each

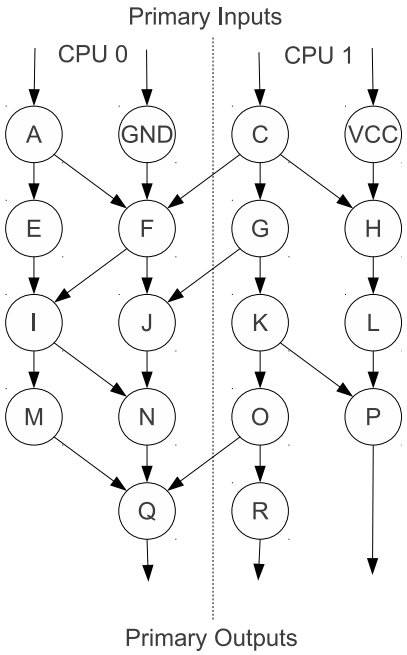

Figure 3. An ideal splitting of the workload may occur in this example if CPU 1 is a little faster than CPU 0.

CPU will complete an equal share of nodes.

But what if CPU 0 gets the head start? Figure 4 illustrates what may occur if CPU 0 completes nodes A, E, and GND while CPU 1 is still computing node C. In this scenario CPU 0 will compute only three nodes out of a total of eighteen.

Because of the potential severity of this problem, we will examine three design features which are intended to reduce the chances that a serious imbalance will develop. All three work by providing a ready supply of additional nodes to any thread that runs out. These techniques are discussed briefly here, and a detailed analysis of their performance is provided in Section 6.

4.2.1 A Shared Master Queue

Rather than dividing the top inputs and constant nodes in a fixed way amongst the processors, a master queue was employed. This allows any CPU which dries up, as CPU 0 did in Figure 4, to pull another primary input or constant node from the master queue and start working from the

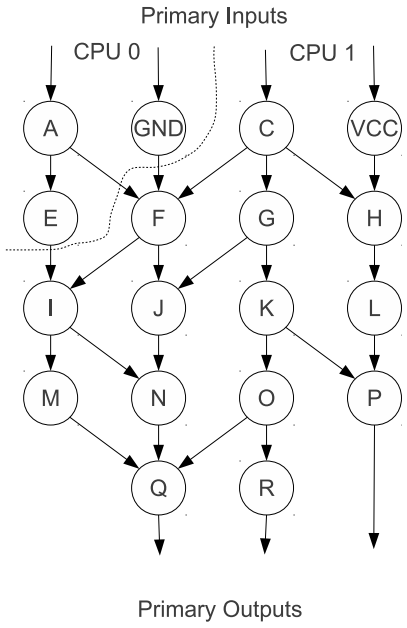

Figure 4. A more pessimistic outcome may result from CPU 0 completing nodes A, E, and GND before CPU 1 is able to complete node C.

top again.

Figure 5 illustrates a possible outcome using this technique. If CPU 1 is about to finish C just after CPU 0 finishes GND, CPU 0 may pull V_{CC} from the master queue, and also complete H, L, and (depending on CPU 1) P. With seven out of eighteen nodes completed by CPU 0, this outcome is substantially better than the one illustrated in Figure 4.

4.2.2 Pre-balancing

Pre-balancing takes advantage of the shared master queue by adding additional high level nodes, specifically those under constant nodes, to the master queue during the initial setup phase. By providing additional re-start points near the top of the netlist, the likelihood that a thread will run out of work is diminished.

When enqueueing nodes during the pre-balancing phase, only nodes which are ready to be computed are added to the master queue. While non-ready nodes might become computable by

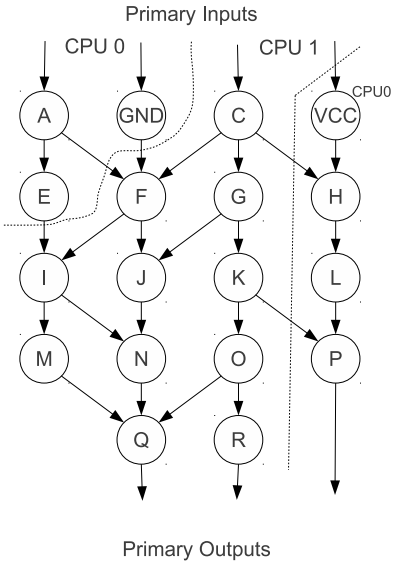

Figure 5. An example of what might occur if CPU 0 finishes A, E, and GND before CPU 1 finishes C, but can then pull V_{CC} from the master queue.

the time they are needed, it is by no means certain that they will be. And since the master queue is a shared resource which requires synchronization, it is used somewhat conservatively.

4.2.3 Re-balancing

Re-balancing refers to a process of randomly assigning additional ready nodes to the master queue during the processing of the queue in hopes of providing stalled threads with additional starting points. While a completed node's children are being placed in the processor's local queue for later computation, a random selection of them are instead enqueued in the master queue.

Random selection is made when the condition

$$(id + r) \mod p = 0$$

is satisfied, where id is the unique identifier of the node, r is the rank of the processor, and p is the number of processors. This distributes the nodes which are selected for master queueing by

each processor equally across all possible nodes.

The re-balancing technique may increase contention on the master queue. Every p^{th} child node discovered by each processor will be passed to the master queue, requiring both the sending and receiving processor to acquire the master queue lock one additional time. As the number of processors increases the number of times each processor acquires the lock decreases proportionally but the number of processors trying to acquire the lock increases.

This means that the locking overhead of re-balancing will be:

$$2 \cdot \left(\frac{n}{p}\right) \cdot p$$

or

$$\mathcal{O}(n)$$

where n is the number of nodes not already in the master queue at the beginning of the parallel section.

The significance of this result with small values of d will largely depend on the length of critical sections. Since they are quite short, consisting only of an enqueue or an empty check followed by a dequeue, n will be multiplied by a relatively small constant factor, and the chance of a collision will be correspondingly small.

Since the run time complexity of re-balancing approaches $T_p(n)$ as p approaches d, as the number of processors increases, its overhead will eventually become significant. Therefore its utility will ultimately depend on its efficacy at preventing imbalances.

5 Implementation

The parallel simulator was implemented in C on GNU/Linux. OpenMP was used for the parallel section and the locks around the master queue. A pseudo-code listing of the parallel simulation cycle is shown in Figure 6 including both pre-balancing and re-balancing techniques.

```
Q_m := the master queue;
netlist := a netlist to be simulated;
cycle := the current cycle;

UpdateFlipFlopsForCycle(netlist,cycle-1);
SetConstantPinValues(netlist,cycle);
EnqueueReadyTopInputNodes(netlist,cycle);
EnqueueReadyConstantPins(Q_m,netlist,cycle);
  // Pre-balancing step
  Also enqueue the ready children of each ready constant pin.

OpenMP parallel segment
{
  p := the number of processors;
  r := the rank of the current processor;
  Q_l := the local queue;

  lock(Q_m);
  while (node := dequeue(Q_m)) {
    unlock(Q_m);
    do {
      if (!complete(node,cycle)) {
        compute(node,cycle);

        while (child := getNextChild(node)) {
          if (
               !child.inQueue
            && isReady(child,cycle)
            && !isComplete(child,cycle)
```

```
            ) {
              child.inQueue := TRUE;

              // Re-balancing condition
              if ((child.unique_id+r)%p) == 0) {
                lock(Q_m);
                enqueue(Q_m,child);
                unlock(Q_m);
              }
              else {
                enqueue(Q_l,child);
              }
            }
          }
        }
        node.inQueue = FALSE;
      } while (node := dequeue(Q_l));
      lock(Q_m);
    }
    unlock(Q_m);
  }
```

Figure 6. The parallel simulate cycle.

The choice of OpenMP makes it easy to derive a sequential algorithm from the parallel one. It is only necessary to remove the local queue, and eliminate the locks surrounding the master queue.

6 Results

In this section we will detail simulation results using the parallel simulator in order to judge its performance relative to the sequential one. As a point of comparison for the benchmark times, Table 1 shows run times for a suite of eleven benchmarks reported by P. S. O'Brien et al. on an Intel Core i3 CPU running at 2.13GHz using 533MHz DDR3 RAM. [6].

Benchmark Name (# of gates)	100 Vector Time (s)
$stereovision2.v(62k)$	24.483
$blob\_merge.v(40k)$	19.277
$boundtop.v(14k)$	8.320
$raygentop.v(10k)$	4.236
$mkDelayWorker32B.v(8.5k)$	5.615
$mkSMAdapter4B.v(4.1k)$	1.619
$or1200.v(3.4k)$	1.439
$diffeq1.v(1.5k)$	0.449
$stereovision3.v(1.5k)$	0.273
$mkPktMerge.v(1.2k)$	0.443
$diffeq2.v(1.1k)$	0.217

Table 1. Simulation time results by P. S. O'Brien et al. [6] Benchmarks with less than two thousand gates were run using one thousand vectors, and times were divided.

6.1 Methodology

Testing was carried out on two parallel computer systems. The first was a typical desktop system with an AMD Phenom X4 quad-core CPU running at 2.60GHz [2]. The second system was a node in the Fundy supercomputing cluster, a big-node cluster which is part of the Atlantic Computational Excellence Network (ACEnet) [1]. The nodes used contained eight dual-core AMD Opteron processors running at 3.2GHz. Both test systems ran 64-bit distributions of GNU/Linux. Times were taken from the beginning initial set-up phase to the implied barrier at the end of the parallel segment. This does not include the time taken to generate test vectors or to write the results.

A set of eleven benchmarks was used, ranging in size from one thousand to sixty-two thousand gates. The larger benchmarks (those with more than two thousand gates) were simulated using

one hundred pseudo-random test vectors. The smaller benchmarks were simulated using one thousand test vectors and their simulation times were then divided by ten. Each set of tests was run at least ten times and the results were averaged over all trials.

Single processor times ($T^*(n)$) were measured using a sequential version of the algorithm, rather than running the parallel algorithm on a single processor. This methodology was selected so that speedup values would reflect the parallel algorithm's performance as compared to the best sequential algorithm [11].

6.2 Load Balancing

Since simulation times will ultimately depend on which load balancing techniques we choose to employ, we will first address their relative performance. As an evaluation criterion for load balance, the proportion of gates completed by the longest thread was chosen. This choice was made due to the fact that gates are the smallest unit of parallelism and are assumed to take roughly equal time to compute.

The desktop system was used exclusively during the comparison of load balancing techniques. All techniques employ the master queue which is initially loaded with all top input and constant nodes.

6.2.1 Master Queue Only

Table 2 establishes a baseline by showing the balance of the algorithm using only the master queue. Notice that in many cases most of the work is done by a single thread. Nevertheless, some of the benchmarks fare reasonably well using only the master queue. Particularly those with fewer total gates, as well as *stereovision*2.*v*. Also interesting is the fact that *stereovision*2.*v* has worse balance with three and four processors than with two.

	Number of Processors			
Benchmark Name (# of gates)	1	2	3	4
$stereovision2.v(62k)$	100.00%	52.31%	59.45%	56.99%
$blob\_merge.v(40k)$	100.00%	98.62%	98.36%	98.40%
$boundtop.v(14k)$	100.00%	84.93%	84.25%	83.48%
$raygentop.v(10k)$	100.00%	71.37%	58.96%	55.77%
$mkDelayWorker32B.v(8.5k)$	100.00%	55.37%	53.71%	50.65%
$mkSMAdapter4B.v(4.1k)$	100.00%	59.42%	58.07%	55.78%
$or1200.v(3.4k)$	100.00%	77.23%	74.86%	73.00%
$diffeq1.v(1.5k)$	100.00%	53.00%	51.69%	47.73%
$stereovision3.v(1.5k)$	100.00%	57.71%	47.81%	45.95%
$mkPktMerge.v(1.2k)$	100.00%	70.89%	60.68%	53.32%
$diffeq2.v(1.1k)$	100.00%	78.36%	65.92%	54.36%
Average	100.00%	69.02%	64.89%	61.40%
Ideal	100.0%	50.0%	$33.\bar{3}$%	25.0%

Table 2. Gates computed by the longest thread on the desktop system using only the master queue.

6.2.2 Pre-balancing

Table 3 shows results using only the pre-balancing technique. These results show substantial improvement over the baseline results, with nearly equal work distribution being achieved in some cases.

There are some notable exceptions. For example the benchmark $diffeq2.v$ does not show a noteworthy improvement over the baseline with this technique.

6.2.3 Re-balancing

Results for the re-balancing technique are shown in Table 4. Some benchmarks clearly benefit more from this technique when compared with the pre-balancing technique. The most noteworthy examples are $diffeq1.v$ and $diffeq2.v$ with four processors. Others such as $blob\_merge.v$ show barely any improvement over the baseline result when using re-balancing.

6.2.4 Combining Pre-balancing and Re-balancing

The results of combining the two techniques are detailed in Table 5. In many cases the two techniques work better together than either does on its own, with imbalances below 36% being

	Number of Processors			
Benchmark Name (# of gates)	1	2	3	4
$stereovision2.v(62k)$	100.00%	51.82%	35.26%	30.14%
$blob\_merge.v(40k)$	100.00%	64.80%	57.85%	56.45%
$boundtop.v(14k)$	100.00%	55.71%	40.66%	32.21%
$raygentop.v(10k)$	100.00%	52.16%	36.58%	29.29%
$mkDelayWorker32B.v(8.5k)$	100.00%	53.10%	42.82%	35.02%
$mkSMAdapter4B.v(4.1k)$	100.00%	53.46%	40.56%	32.06%
$or1200.v(3.4k)$	100.00%	52.74%	37.56%	28.42%
$diffeq1.v(1.5k)$	100.00%	54.57%	41.94%	37.49%
$stereovision3.v(1.5k)$	100.00%	53.28%	35.51%	27.48%
$mkPktMerge.v(1.2k)$	100.00%	52.13%	35.37%	29.50%
$diffeq2.v(1.1k)$	100.00%	59.57%	48.78%	42.11%
Average	100.00%	54.85%	41.17%	34.56%
Ideal	100.0%	50.0%	$33.\bar{3}\%$	25.0%

Table 3. Gates computed by the longest thread on the desktop system using only pre-balancing

achieved with every benchmark except $diffeq2.v$ using four processors.

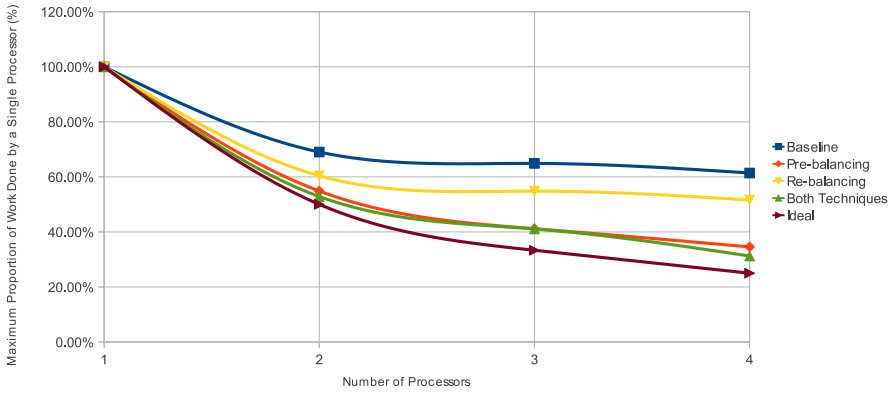

Figure 7. Comparison of load balancing techniques averaged over all benchmarks. Lower is better.

Figure 7 compares the performance of the two load balancing techniques against the baseline (master queue only) results. The results are also compared against an idealized (perfectly balanced) algorithm averaged over all benchmarks. The average proportion of work done by a single processor is shown along the y-axis, with the number of processors employed being shown along the x-axis.

The master queue alone clearly provides some degree of balancing, but becomes less effective as the number of processors increases. We can see that re-balancing alone proves inferior to

	Number of Processors			
Benchmark Name (# of gates)	1	2	3	4
$stereovision2.v(62k)$	100.00%	52.25%	35.02%	28.40%
$blob\_merge.v(40k)$	100.00%	93.08%	96.76%	93.71%
$boundtop.v(14k)$	100.00%	67.21%	82.31%	79.91%
$raygentop.v(10k)$	100.00%	54.01%	37.80%	45.66%
$mkDelayWorker32B.v(8.5k)$	100.00%	57.27%	43.16%	46.32%
$mkSMAdapter4B.v(4.1k)$	100.00%	55.01%	38.89%	50.40%
$or1200.v(3.4k)$	100.00%	61.24%	67.78%	68.65%
$diffeq1.v(1.5k)$	100.00%	52.97%	63.25%	34.55%
$stereovision3.v(1.5k)$	100.00%	53.84%	39.13%	39.83%
$mkPktMerge.v(1.2k)$	100.00%	61.08%	47.71%	45.76%
$diffeq2.v(1.1k)$	100.00%	56.15%	51.07%	34.10%
Average	100.00%	60.37%	54.81%	51.57%
Ideal	100.0%	50.0%	33.$\bar{3}$%	25.0%

Table 4. Gates computed by the longest thread on the desktop system using only re-balancing

pre-balancing when averaged over all benchmarks.

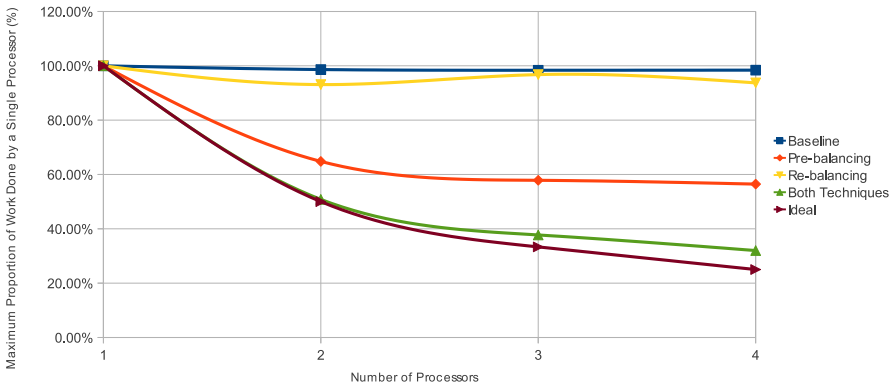

Figure 8. Comparison of load balancing techniques for blob_merge.v, a 40,000 gate circuit. Lower is better.

Figure 8 makes the same comparison using $blob\_merge.v$. Here, the master queue is nearly useless in balancing the load. There is also a stark contrast between the effectiveness of re-balancing when contrasted with pre-balancing. When combined, the two techniques perform substantially better than either on its own, giving a balance at four processors of 32%, only 7% away from ideal.

Finally, Figure 9 compares both techniques using the smallest circuit, $diffeq2.v$. Here the mas-

	Number of Processors			
Benchmark Name (# of gates)	1	2	3	4
$stereovision2.v(62k)$	100.00%	51.42%	34.46%	29.25%
$blob\_merge.v(40k)$	100.00%	50.82%	37.73%	31.99%
$boundtop.v(14k)$	100.00%	53.46%	42.79%	31.69%
$raygentop.v(10k)$	100.00%	51.50%	35.80%	29.68%
$mkDelayWorker32B.v(8.5k)$	100.00%	54.63%	40.74%	32.79%
$mkSMAdapter4B.v(4.1k)$	100.00%	54.00%	37.77%	29.41%
$or1200.v(3.4k)$	100.00%	52.09%	37.19%	28.76%
$diffeq1.v(1.5k)$	100.00%	53.24%	61.82%	34.20%
$stereovision3.v(1.5k)$	100.00%	52.48%	36.65%	27.67%
$mkPktMerge.v(1.2k)$	100.00%	51.74%	35.52%	28.49%
$diffeq2.v(1.1k)$	100.00%	55.93%	51.41%	39.80%
Average	100.00%	52.85%	41.08%	31.25%
Ideal	100.0%	50.0%	$33.\bar{3}\%$	25.0%

Table 5. Gates computed by the longest thread on the desktop system using re-balancing and pre-balancing

ter queue needs much less assistance as the load is moderately balanced before either of the other two techniques are employed. The addition of pre-balancing provides a noticible improvement. With the addition of re-balancing we see a somewhat better result than with pre-balancing using four processors. However when the two techniques are combined, the result is worse than with re-balancing alone.

Two patterns emerge here. First, smaller circuits generally have superior baseline performance, in that the master queue performs much better without the assistance of additional load balancing measures. Secondly, pre-balancing tends to perform better with the larger circuits than re-balancing, again with the exception of $stereovision2.v$.

A possible explanation for the improved baseline with smaller circuits may be that the depth of a circuit - or the distance from a primary input to a primary output - is generally shorter. This would leave less room for a substantial imbalance to grow. In cases where a thread goes dry, there may be enough additional primary inputs and constant nodes in the master queue to begin to make up for it, rendering pre-balancing superfluous.

Table 6 provides maximum depths for each benchmark circuit. These figures correlate well

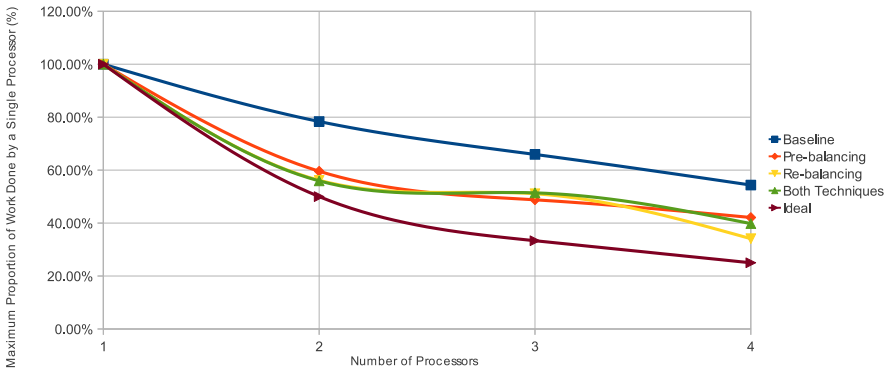

Figure 9. Comparison of load balancing techniques for diffeq2.v, a 1.1 thousand gate circuit. Lower is better.

to the master queue only balances shown in Table 2, supporting the hypothesis that shallower circuits balance better than deeper ones without the aid of additional load balancing measures.

Benchmark Name	Maximum Depth
$stereovision2.v(62k)$	104
$blob_merge.v(40k)$	207
$boundtop.v(14k)$	518
$raygentop.v(10k)$	214
$mkDelayWorker32B.v(8.5k)$	191
$mkSMAdapter4B.v(4.1k)$	164
$or1200.v(3.4k)$	945
$diffeq1.v(1.5k)$	12
$stereovision3.v(1.5k)$	48
$mkPktMerge.v(1.2k)$	48
$diffeq2.v(1.1k)$	196

Table 6. Maximum depth (or the maximum distance from a primary input or constant node to a primary output node) for each benchmark.

It is also possible that some circuits, especially smaller ones, simply have fewer nodes under top level nodes, and that the constant nodes have lower fanout. This may explain why re-balancing works better with some circuits where the additional starting points can come from anywhere in the circuit and do not depend on the degree of fanout of the constant nodes. Which leads us to a possible explanation of the second observation.

With larger and presumably deeper circuits there may be more opportunity for a thread to dry up. This may mean that primary inputs and constant nodes provide an insufficient number of restart points. Larger circuits may also have a greater degree of fanout from the constant nodes leading to more second generation nodes for pre-balancing to use. Therefore the initial high-level nodes provided by pre-balancing may be both more numerous and have a greater pay-off than with a smaller circuit. In this case re-balancing may have a reduced pay off due to the reduced likelihood that its added restart points will be either timely or near the top of the netlist.

Because of the similarity of the pre-balancing result with the combined result, and due to the added overhead of re-balancing, simulation times were used to make the determination as to whether or not to include re-balancing in the final implementation.

	Number of Processors			
Benchmark Name	1	2	3	4
$stereovision2.v(62k)$	13.262	9.844	7.708	7.073
$blob\_merge.v(40k)$	11.634	7.750	5.512	5.110
$boundtop.v(14k)$	4.417	3.105	2.627	2.358
$raygentop.v(10k)$	2.258	1.743	1.347	1.226
$mkDelayWorker32B.v(8.5k)$	2.847	2.217	1.948	1.904
$mkSMAdapter4B.v(4.1k)$	0.889	0.720	0.605	0.638
$or1200.v(3.4k)$	0.844	0.746	0.670	0.685
$diffeq1.v(1.5k)$	0.139	0.132	0.152	0.118
$stereovision3.v(1.5k)$	0.153	0.133	0.111	0.132
$mkPktMerge.v(1.2k)$	0.206	0.214	0.185	0.205
$diffeq2.v(1.1k)$	0.066	0.065	0.071	0.064

Table 7. Benchmark simulation times on an AMD Phenom X4 @ 2.60GHz using both load balancing techniques. Times are given in seconds for 100 pseudo-random test vectors. Benchmarks with less than 2000 gates were run with 1000 vectors and times were divided accordingly.

Table 7 details simulation times using both techniques. From these times, we get the following average speedups for each processor:

$$(1.00, 1.22, 1.43, 1.49).$$

Likewise, Table 8 details simulation times using only pre-balancing. Performing the same

calculation there, we get:

$$(1.00, 1.41, 1.62, 1.49).$$

We can see that the two processor result shows the largest difference, and that for each number of processors, pre-balancing only shows equal or better average speedup.

This tells us two things. Firstly, that using only pre-balancing is likely to give us better overall speedup. Secondly, and more importantly, it demonstrates that by using only pre-balancing, we get that speedup with fewer processors, lowering the cost of the algorithm. Therefore the choice was made to use only pre-balancing in the final implementation.

	Number of Processors			
Benchmark Name	1	2	3	4
$stereovision2.v(62k)$	13.261	6.974	5.849	5.736
$blob\_merge.v(40k)$	11.555	8.283	7.325	7.364
$boundtop.v(14k)$	4.426	2.716	2.315	2.223
$raygentop.v(10k)$	2.314	1.389	1.197	1.244
$mkDelayWorker32B.v(8.5k)$	2.859	2.086	1.931	1.985
$mkSMAdapter4B.v(4.1k)$	0.888	0.648	0.583	0.575
$or1200.v(3.4k)$	0.843	0.684	0.618	0.703
$diffeq1.v(1.5k)$	0.142	0.104	0.087	0.110
$stereovision3.v(1.5k)$	0.152	0.125	0.103	0.142
$mkPktMerge.v(1.2k)$	0.208	0.195	0.163	0.206
$diffeq2.v(1.1k)$	0.069	0.051	0.051	0.061

Table 8. Benchmark simulation times on an AMD Phenom X4 @ 2.60GHz. Times are given in seconds for 100 pseudo-random test vectors. Benchmarks with less than 2000 gates were run with 1000 vectors and times were divided accordingly.

6.3 Results Using the Desktop System

Since most research and development on Odin II is done using desktop workstations, we'll focus first on the performance of the parallel simulator when using typical desktop hardware. Table 8 details the simulation times of the algorithm on desktop hardware using up to four processors.

Figure 10 plots speedup on the desktop system with up to four processors. The largest circuit,

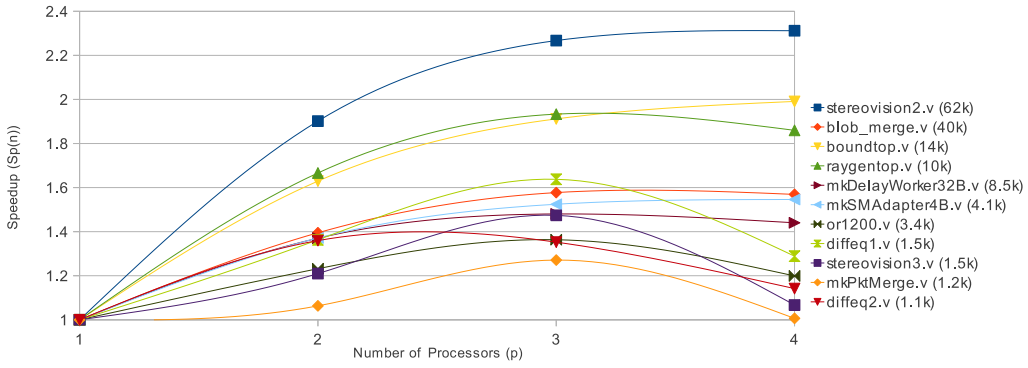

Figure 10. Speedup vs. number of processors for on a Phenom X4 @ 2.60GHz.

*stereovision*2.*v* shows the most significant improvement in simulation time with the addition of processors, reaching a peak speedup of 2.6 using four processors. In second place is is *boundtop.v* which hits a peak speedup of 2 with four processors. Also visible is a significant levelling off of the speedup curves between three and four processors for most benchmarks, with several smaller benchmarks elbowing out between two and three processors.

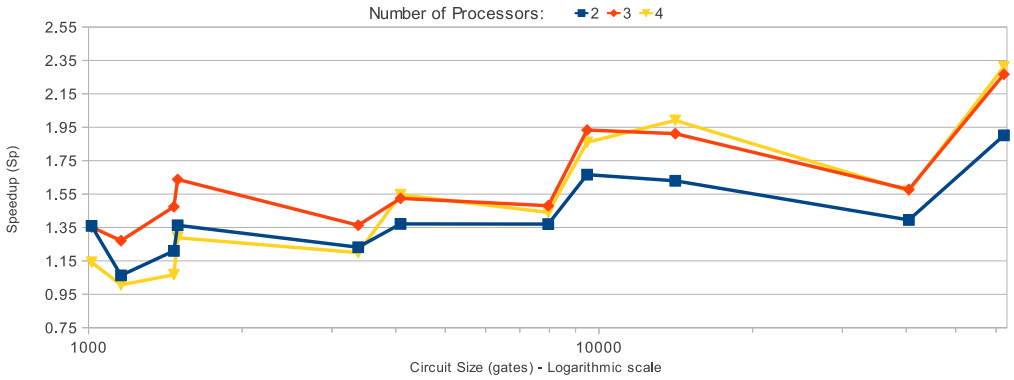

Figure 11. Speedup vs. problem size on a Phenom X4 @ 2.60GHz.

To further explore the relationship between circuit size and speedup, Figure 11 plots the circuit size against speedup for up to four processors. A noticeable correlation can be observed, with the largest circuits showing a trend towards better speedup. Also interesting is the fact that smaller circuits tend to elbow out between three and four processors, whereas larger circuits tend to speedup slightly better with four processors.

One exception to the rule is *blob_merge.v* which speeds up slower than other benchmarks of similar size. We saw in Figure 8 that this benchmark tends to balance poorly using only pre-balancing. Table 3 shows that this is especially true when compared to the other benchmarks. This may explain why *blob_merge.v* displays unusually poor parallel speedup for its size.

6.4 Results Using Fundy

Results for up to eight processors were obtained using the Fundy cluster [1]. Despite the fact that most work on Odin II is done using desktop hardware, it is useful to examine how our algorithm scales to a system with larger numbers of processors.

	Number of Processors							
Benchmark Name	1	2	3	4	5	6	7	8
$stereovision2.v(62k)$	26.316	12.292	10.318	9.786	9.803	10.226	10.610	10.821
$blob_merge.v(40k)$	23.567	15.889	14.603	13.530	12.467	12.589	13.054	12.528
$boundtop.v(14k)$	8.865	5.641	4.772	4.414	4.408	4.459	4.296	4.361
$raygentop.v(10k)$	4.724	2.669	2.197	2.060	1.958	1.950	1.978	2.009
$mkDelayWorker32B.v(8.5k)$	5.616	4.065	3.570	3.498	3.463	3.578	3.530	3.668
$mkSMAdapter4B.v(4.1k)$	1.744	1.283	1.136	1.043	1.037	1.049	1.041	1.080
$or1200.v(3.4k)$	1.666	1.354	1.169	1.172	1.162	1.128	1.176	1.259
$diffeq1.v(1.5k)$	0.264	0.205	0.165	0.164	0.177	0.189	0.180	0.190
$stereovision3.v(1.5k)$	0.288	0.282	0.243	0.240	0.243	0.259	0.273	0.289
$mkPktMerge.v(1.2k)$	0.357	0.372	0.341	0.322	0.312	0.330	0.345	0.359
$diffeq2.v(1.1k)$	0.079	0.079	0.072	0.072	0.074	0.076	0.077	0.077

Table 9. Benchmark simulation times on Fundy, given in seconds. Benchmarks with less than 2000 gates were run using 1000 vectors and their simulation times were divided by 10.

Table 9 details average simulation times on Fundy with up to eight processors while Figure 12 plots the corresponding speedup calculations. At first glance we notice that Fundy's 3.2GHz Opterons take roughly twice as long to execute the same benchmark as the desktop system's Phenom, which runs at only 2.6GHz. Fundy uses older 8000-series Opterons, which are based on the 90nm K8 microarchitecture, while the desktop system's Phenom is based on the significantly faster 65nm K10 microarchitecture [1] [9].

Fundy also uses slower 667MHz DDR2 RAM when compared to the desktop system's 800MHz

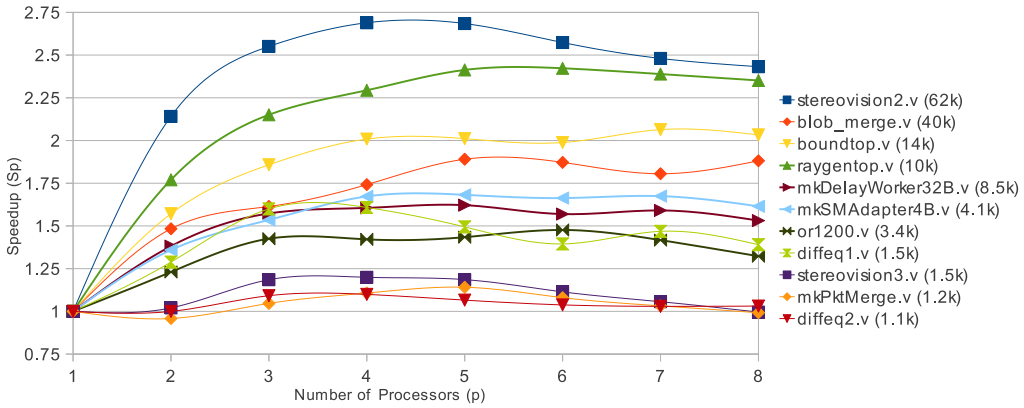

Figure 12. Speedup vs. number of processors for on Fundy.

DDR2 [1]. When compared to the results in Table 1 which were acquired using a system with 533MHz DDR3 we see that those times are also roughly double those of the desktop system. This may point to a memory bottleneck as a likely cause of this difference.

The largest circuit shows a steady falloff in simulation time from 26.3 seconds to a minimum of 9.8 seconds as the number of processors increases from one to four. Like the desktop results, the smaller circuits show a slower falloff in simulation time with additional processors. We can also see that our smaller circuits tend to elbow out earlier than the larger ones.

*Stereovision*2.*v* shows a significantly better speedup than the other benchmarks, peaking at 2.7 times sequential speed with five processors, where a pronounced elbowing out can also be observed. The two processor result for *stereovision*2.*v* shows evidence of super-linear speedup, with a speedup of 2.14. This is probably the result of an increase in cache hit rate caused by the efficient division of work across two processor caches.

7 Discussion

Given an ideal circuit which can be cleanly divided into p independent workloads and an idealized parallel computer, we can expect the linear speedup with this algorithm which was calculated in Section 3. However, with real world circuits and real world computers, we get somewhat different results. Focusing on the Fundy speedup result for $stereovision2.v$ shown in Figure 12, we can enumerate several candidate explanations for the elbowing out of the algorithm.

Benchmark Name	Average Setup Time (s)
$stereovision2.v(62k)$	$3.86x10^{-2}$
$blob_merge.v(40k)$	$1.99x10^{-2}$
$boundtop.v(14k)$	$1.88x10^{-2}$
$raygentop.v(10k)$	$9.58x10^{-3}$
$mkDelayWorker32B.v(8.5k)$	$2.12x10^{-2}$
$mkSMAdapter4B.v(4.1k)$	$5.60x10^{-3}$
$or1200.v(3.4k)$	$7.07x10^{-3}$
$diffeq1.v(1.5k)$	$2.82x10^{-4}$
$stereovision3.v(1.5k)$	$6.93x10^{-4}$
$mkPktMerge.v(1.2k)$	$1.35x10^{-3}$
$diffeq2.v(1.1k)$	$1.08x10^{-4}$

Table 10. Average setup times for each benchmark on Fundy, corresponding to 100 setups.

First we'll look at the basic design. We know that this algorithm has a sequential setup phase which is not amenable to parallelism. As the parallel queue processing phase speeds up, we can expect this inherently sequential portion to remain relatively constant. Table 10 details setup times for each benchmark. Comparing these values to the simulation times in Table 9 reveals that they are orders of magnitude lower. They are therefore negligible.

Another likely candidate is the time taken to spawn threads. It is reasonable to assume that as the number of threads spawned increases, that the amount of time consumed by the program constructing these threads will increase. Since thread creation usually requires a system call, it can be assumed that this overhead is not distributed across the processors and ultimately adds to the overall run time of the program.

The total thread spawn times for two through eight processors averaged across all benchmarks

are:

$$(6.11x10^{-6}, 7.30x10^{-6}, 7.75x10^{-6}, 8.13x10^{-6}, 8.05x10^{-6}, 8.85x10^{-6}, 9.99x10^{-6})$$

which again are orders of magnitude lower than our simulation times. They can therefore be neglected.

	Number of Processors							
Benchmark Name	1	2	3	4	5	6	7	8
stereovision2.v(62*k*)	100.00%	53.04%	38.19%	30.08%	24.00%	19.67%	16.68%	15.69%
blob_merge.v(40*k*)	100.00%	66.28%	60.42%	57.10%	54.89%	52.91%	52.90%	48.23%
boundtop.v(14*k*)	100.00%	55.78%	41.99%	33.59%	28.82%	25.71%	24.16%	23.03%
raygentop.v(10*k*)	100.00%	53.51%	38.53%	30.97%	25.46%	21.73%	19.44%	17.31%
mkDelayWorker32B.v(8.5*k*)	100.00%	58.20%	46.89%	37.25%	31.15%	27.05%	23.75%	22.25%
mkSMAdapter4B.v(4.1*k*)	100.00%	53.49%	41.14%	32.39%	27.38%	24.15%	22.54%	20.71%
*or*1200.*v*(3.4*k*)	100.00%	53.21%	39.94%	29.95%	24.84%	20.28%	17.60%	15.66%
*diffeq*1.*v*(1.5*k*)	100.00%	53.80%	42.80%	40.27%	42.64%	42.65%	42.54%	42.09%
stereovision3.v(1.5*k*)	100.00%	53.40%	36.28%	28.57%	24.17%	20.96%	18.93%	17.85%
mkPktMerge.v(1.2*k*)	100.00%	54.20%	37.89%	30.41%	26.22%	23.57%	21.30%	19.76%
*diffeq*2.*v*(1.1*k*)	100.00%	64.61%	48.97%	43.31%	43.32%	42.85%	43.78%	47.16%

Table 11. Gates computed by one processor on Fundy using pre-balancing. Averaged over 10 simulations of 100 vectors. Those with less than 2000 gates were simulated using 1000 vectors.

Another obvious suspect is the increasing load imbalance which was highlighted earlier. Load imbalance on Fundy was measured and the results are detailed in Table 11. Idle times were calculated by comparing these imbalance figures with their idealized counterparts, and the resulting corrections were applied to the values in Table 9. Corrected timings and speedup values were calculated accordingly and results for the five largest benchmarks are displayed in Figure 13.

This provides us with a theoretical result showing how the algorithm should behave in the absence of any idle time. There is clearly an overall increase in speedup affecting all results. Speedup increases to 2.83 with *stereovision.v* using four processors. Notably absent from the corrected results thus far is any shift in the locations or presence of the pronounced elbows affecting the larger circuits between three and five processors.

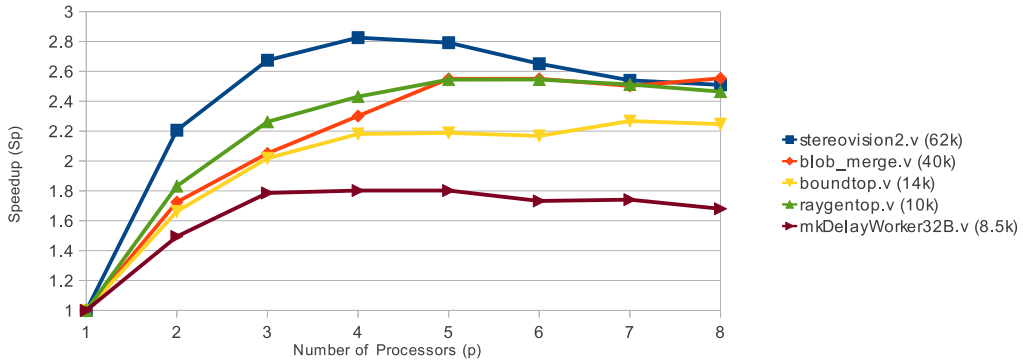

Figure 13. Speedup vs. number of processors on Fundy. Corrected for load imbalance.

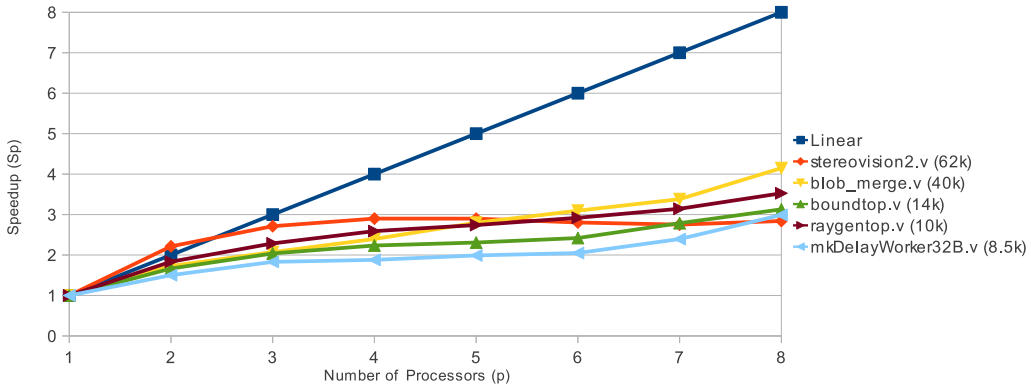

Figure 14. Speedup vs. number of processors on Fundy. Corrected for load imbalance, and approximate wait times.

Next we account for the fact that all threads, for at least some of their run time, must co-ordinate around the master queue. Wait times corresponding to the times in Table 9 were measured. When averaged over several simulations and integrated into the corrections above, we arrive at Figure 14 which depicts the theoretical behaviour of the algorithm on Fundy in the absence of both queue waiting time and idle time.

The corrected results show a clear indication of a bottleneck, preventing the algorithm from speeding up above about three or four times sequential speed. The most likely culprit for this bottleneck is memory. With increasing numbers of processors accessing main memory, eventually the memory bus becomes saturated and cannot supply data fast enough to the processors.

8 Summary

We described the design and implementation of a parallel netlist simulator for Odin II using C and OpenMP. We have described the simulator's design in Sections 3 and 4 and its implementation in Section 5. It has been shown in Section 6 to provide good speedup on typical desktop hardware with up to four processors and on server hardware with up to five processors. We have also demonstrated that this algorithm performs even better relative to the sequential version as the circuit size increases.

It has also been demonstrated in Section 7 that along with synchronization and idle time, memory may be a significant bottleneck in achieving better performance with large numbers of processors.

9 Conclusion

The parallel netlist simulator provides significant speedup using small numbers of processors. Speedup is generally better with large circuits, but also depends on the internal structure of the individual circuit. Results point to idle time, synchronization overhead, and memory bandwidth as major limiting factors in scaling performance to larger numbers of processors.

References

[1] ACEnet. The Fundy Cluster, C/O The ACEnet Wiki, https://wiki.ace-net.ca/index.php/Fundy. 2011.

[2] AMD. AMD Product Page for the Phenom X4 9950 CPU, http://products.amd.com/(S(i0cnmd452yr2vabpaegkum45))/pages/DesktopCPUDetail.aspx?id=447. 2011.

[3] Berkeley. The ABC Homepage, http://www.eecs.berkeley.edu/ alanmi/abc. 2011.

[4] Google Code. Odin II Project Page, https://code.google.com/p/odin-ii/. 2011.

[5] P. Jamieson, K. B. Kent, F. Gharibian, and L. Shannon. Odin II - An Open-Source Verilog HDL Synthesis Tool for CAD Research. In *Proceedings of the 2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '10, pages 149–156, Washington, DC, USA, 2010. IEEE Computer Society.

[6] P. S. O'Brien, A. Furrow, J. Libby, and K. B. Kent. A Simple Tractable Approach to Design Tool Verification Through Simulation and Statistics, To Appear in IEEE Conference on Field Programmable Technologies (6 pages). New Delhi, India, December 12-14, 2011.

[7] U. of Toronto. The T-VPack Homepage, http://www.eecg.toronto.edu/~vaughn/vpr/vpr.html. 2011.

[8] OpenMP.org. About OpenMP ARB and OpenMP.org, http://openmp.org/wp/about-openmp/. 2011.

[9] Oracle.com. SunFire X4600 M2 CPU Support, http://download.oracle.com/docs/cd/E19121-01/sf.x4600/819-5040-11/migchap.html. 2011.

[10] T. Rauber and G. Runger. Introduction. In *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2010.

[11] T. Rauber and G. Runger. Performance analysis of parallel programs. In *Parallel Programming: for Multicore and Cluster Systems*. Springer-Verlag Berlin Heidelberg, 2010.