

A Guided Tour of SimCluster

Application Note #4
Chris Browy
February 2007

SimCluster is an innovative parallel, distributed simulation environment that provides a scalable, open, and flexible solution to increase RTL and gate-level simulation performance and capacity by 300-700% or more. SimCluster supports Verilog and VHDL design methodologies and the most popular simulators (NC-Verilog, Verilog-XL, VCS, and ModelSim), hardware accelerators, and emulators.

SimCluster supports popular computing solutions including compute clusters and symmetric multiprocessors (SMP) involving from 2 to 10s of processors. The SimCluster Communication Protocol (SCP) uses the ubiquitous TCP/IP sockets over LANs and Unix sockets on SMPs to tune performance. SimCluster also supports Platform LSF for job queuing of SimCluster runs.

The application note will provide a step-by-step guide to using SimCluster including tool overview, how to partition a design, configuration and set-up, and simulation. A simple computer design example is used to illustrate these steps. A complete evaluation kit is available from Avery through your local sales representative.

NOTE: This application note requires VCS 7.1.1 or later, and NC-Sim 4.1 or later.

Design Overview

The simple computer (Figure 1) has a processor, cache, and memory. The processor is designed to fetch data from the cache using load/store instructions. A functional test program is hard-coded in the processor and verifies the computer load/store behavior including write-through cache operation and cache read hits and misses.

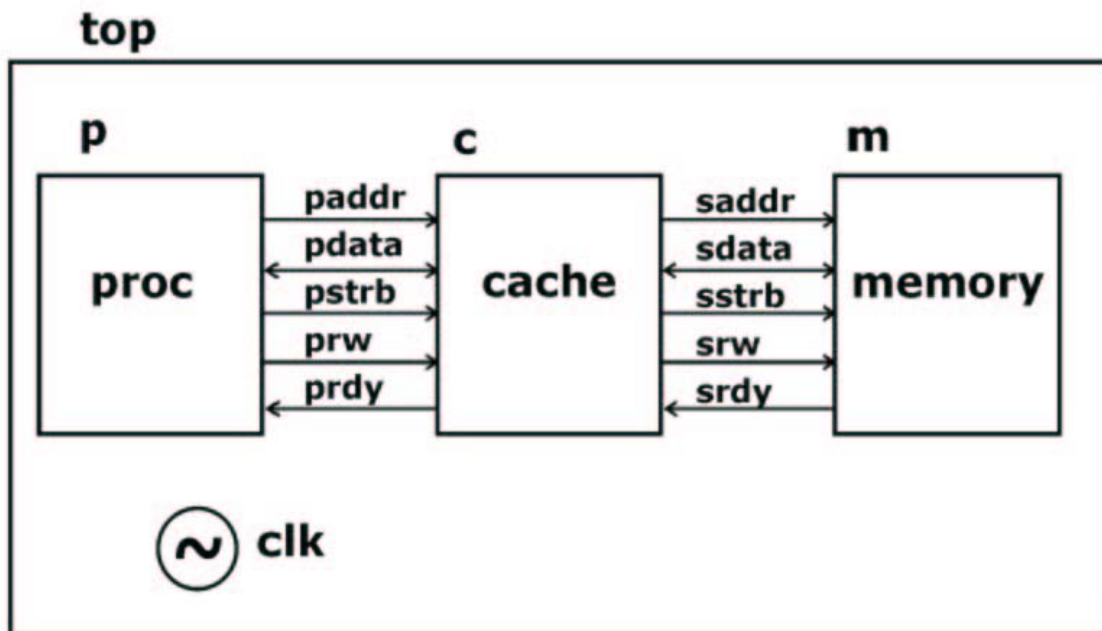


Figure 1. Simple computer logical block diagram

The simple computer design is implemented in 2 versions: a) Verilog only, <install_dir>/src/verilog, and b)

mixed Verilog and VHDL version, <install_dir>/src/mixed. The design files follow the logic block diagram:

```
top.v
proc.v
cache.v
set.v, set.vhd
memory.v
```

SimCluster Overview

SimCluster performs a single logical simulation using multiple tightly-synchronized, independent simulators running in parallel. SimCluster manager coordinates simulation of the top-level module of the design and interfaces with the multiple simulators and independent C/C++ processes that simulate each of the partitions. SimCluster manager controls all simulators in the cluster performing:

1. Global initialization
2. Time stepping and synchronization
3. Simulation value propagation on ports and variables
4. Job control (interrupt for interactive debug, termination)

SimCluster provides several functions to configure and control parallel simulation which are summarized below. These functions are typically added to Verilog/VHDL module descriptions to configure each simulation partition.

```
module_dynamic_link    Declares the dynamic modules (SimCluster manager only)
module_dynamic_link_double

master_channel_total  Declares connection port number (SimCluster manager only)
channel_connect       Connects a dynamic module partition to SimCluster manager
channel_instance      Retrieves full hierarchical pathname of instance
channel_cfunc         Calls a remote C function
channel_task          Calls a remote task or procedure
channel_variable      Reads value of another dynamic module's variables/records
channel_variable_mirror,
channel_variable_map  Sets up auto copy-on-change of another dynamic module's variables

channel_variable_set  Writes another dynamic module's variables/records
channel_port_disable  Disables propagating port value changes between partitions
channel_clock         Defines clock ports of dynamic module simulation partitions
next_sync            Defines next SimCluster inter-simulator synchronization time
sync_on,
sync_on_always       Defines next SimCluster inter-simulator synchronization time
socket_traffic_bucket Monitors inter-simulator messages
```

Guidelines on Partitioning a Design

There are several factors that must be considered for the effective use of SimCluster including the a) compute environment, b) design and test bench, and c) current simulation runtime metrics. Each of these factors impose practical constraints on how best to implement a simulation cluster. Let's start by considering the compute environment. SimCluster provides the best performance when simulation partitions communicate over the fastest links available.

1. What is the capacity of each simulator/computing platform? What is the estimate of the largest RTL, gate-level, or mixed-level model that can be run in uni-processor mode?
2. How many network ports are available on the subnet (LAN) and what speed are they?
3. How many processors in SMPs?

Rule #1: You should not have more simulation partitions than you have processors on a subnet switch

The design and test bench attributes should be considered next. These involve structural attributes:

1. What is the hierarchical structure of the design model and test bench (top-level)? Understanding the hierarchy is an important starting point to partitioning.

Guideline #1: Partitioning is easier if done on the existing hierarchy of the test bench/design (Figure 2)

2. What is the gate count estimate/actual in each module? Modules that contain roughly the same numbers of gates represent good candidates for partitioning. Sometimes the partitions are formed at different levels of the design hierarchy to balance the relative size of partitions. Other times there is a distinct lack of hierarchy as illustrated in Figure 3. In a situation like this, it is advantageous to add a new pseudo-top-level. Each partition is a copy of the original top-level but with only a semi-exclusive subset of the instances. It would be common to duplicate some test bench and environment modules.

Guideline #2: New pseudo-hierarchy may be necessary if there is insufficient hierarchy (Figure 3).

3. What is the clock distribution scheme? It is better to partition the design on its slower interface boundaries. This reduces the number of times the simulators need to sync-up.

Guideline #3: Partitioning should be done to isolate clock domains in partitions and minimize clock propagation

4. Is there PLI or FLI code? Can this PLI/FLI code operate in a replicated or distributed fashion? In most cases, PLI/FLI routines implement test bench extensions and can therefore be replicated in all simulation partitions. However there are special cases, different PLI/FLI routines are dependent on program state and access global shared variables. In this case, SimCluster's remote procedure call or remote memory access capabilities will be required.

5. Do the Verilog/VHDL test benches or bus functional models use hierarchical object references? Hierarchical or cross module references to ports and variables require special handling if they span simulation partitions. Partitioning decisions should focus on minimizing cross module references wherever possible.

Guideline #4: Partitioning should minimize hierarchical object references (usually limited to test benches)

Rule #2: Testbench and design hierarchy imposes limits on number of simulation partitions

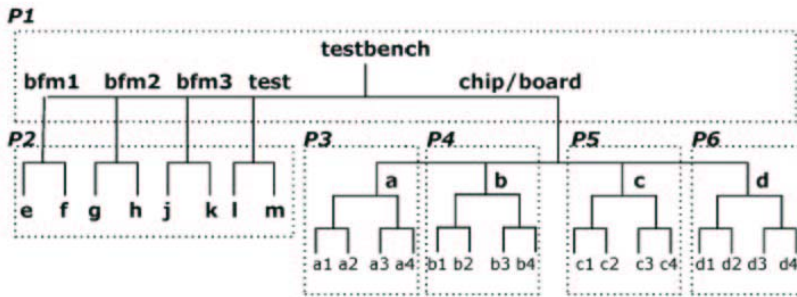
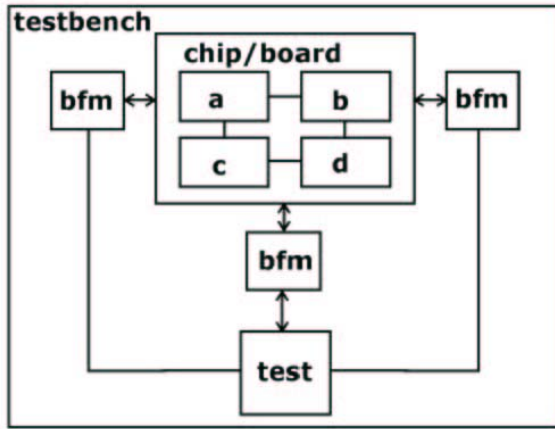


Figure 2. Simple partitioning follows test bench and design hierarchy

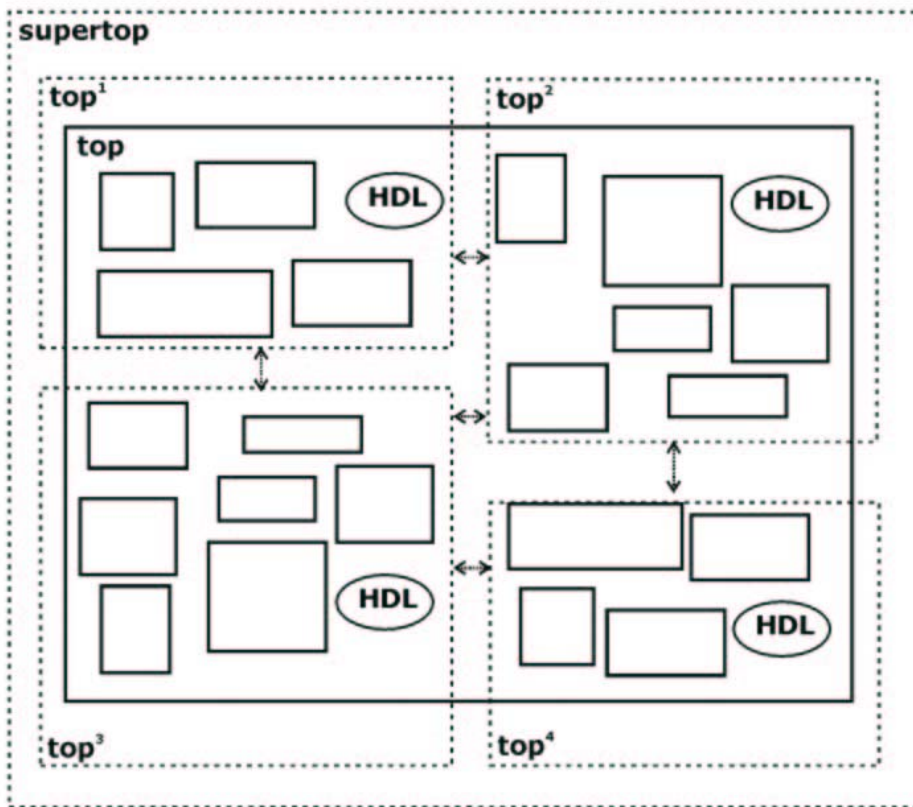


Figure 3. Pseudo-hierarchy is added above top-level, super top, to facilitate partitioning flat designs

Finally, single-process simulation workload metrics need to be analyzed to understand the potential of distributing the workload among multiple partitions in the parallel simulation environment. The basic theory of parallel computing applications dictates that the ideal performance speedup factor, S , is in a linear relationship to the number of partitions, N , when a) the CPU workload of each of the partitions, W_n is roughly equal, and b) the sum of the workloads, W_n , equals the single process workload, W_s .

$$W_s = \sum W_n$$

When either of these rules are deviated, performance speedups will be less than ideal. For instance, if any partition's CPU workload greatly exceeds that of the others', the overall performance will be governed by that one partition. A key simulation workload metric that can be used to guide partitioning decisions is the simulation runtime performance (ie. rate of work) measured in *cycles per second* (cps) as defined as:

```
cps = (simulated time / min clock period)/wall clock time

where,
simulated time = length of simulation (sec)
min clock period = period of highest frequency clock (seconds/cycle)
wall clock time = real time to complete simulation (sec)
```

The single-process simulations leading up to the SimCluster simulations may provide key cps metrics on the sub-units that may be considered for simulation partitions. Other measures which can be used as estimates of simulation workload include simulated events, toggle count (gate-level only), power analysis, and code coverage metrics. Some of these metrics can be generated in a hierarchical report format which is directly applicable to determining the allocation of sub-units to one or more partitions.

Guideline #5: Gathering metrics of the sub-units in a design can provide key grouping information

Finally, to calculate an upper bound on the number of 'simulatable' partitions that is practical, the SimCluster's SCP overhead must be taken into account. To realize good overall performance speedups, experience has shown that the SCP overhead should be kept below 30% of the parallel simulation CPU overhead in each of the parallel simulations runs. The SCP overhead originates from inter-process communications latency of the simulation cluster which is implemented using either a computer network or SMP computer. An upper-bound on the number of partitions based on cps metrics can be formulated based on the analyzing the switching frequency of the signals between the simulation partitions, otherwise known as interface frequency. Another guideline can be postulated by revising the cps formula to use the interface frequency instead of maximum frequency of the entire design.

```
cpsinterface clock = (simulated time / interface clock period)/wall clock time

where,
interface clock period = period of highest frequency interface clock
```

Guideline #6: A practical upper limit on the number of partitions (N) to target is:

$$N^2 * cps_{interface\ clock} < 7,500 \text{ (1 Ghz CPU, 1 Gigabit Ethernet)}$$

$$N^2 * cps_{interface\ clock} < 15,000 \text{ (SMP, 1GHz)}$$

Multiple SimCluster configurations should be defined and implemented which can be selected at runtime based on the number of compute resources available.

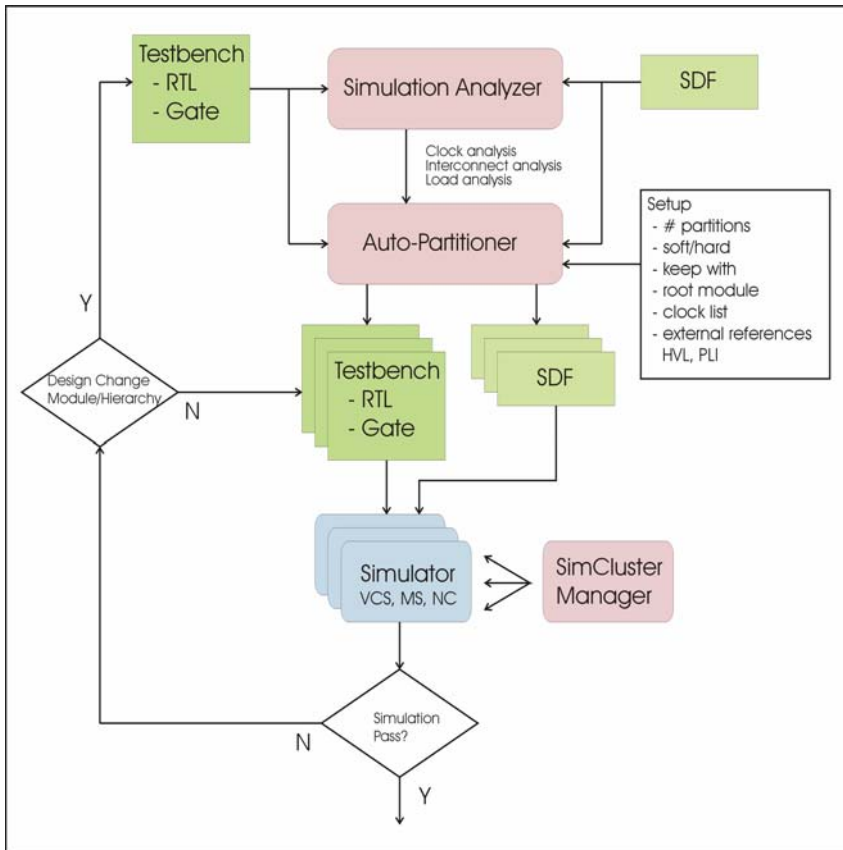


Figure 4: SimCluster supports automated design analysis and partitioning

Partitioning the Simple Computer Design

As Figure 4 illustrates that SimCluster offers simulation evaluation tools which can run in the single process simulation environment. These tools can generate valuable information on design/ sub-module workloads, inter connectivity between blocks, clock analysis, and design hierarchy report. Analysis of the report(s) generated by these evaluation tools can assist in coming up with a well balanced seed for the partitioner, thus leading to optimal speedup by balancing the workload and minimizing the interconnects between the partitions. Usage of the evaluation tools is documented in `evaltool/README`. In this example we ran the single process simulation with `design_hierarchy_report`, `interconnect_analysis` and `vck_eval`, under the `sim/vcs` directory using the `run_single_process.sh` script, by adding the following initial block to the `auto_part_rtl/single_sim.v` file:

```
initial
begin
  $design_hierarchy_report(2, 1);
  $interconnect_analysis(top);
  $vck_eval(1, 50, 2);
end
```

`design_hierarchy_report` and `vck_eval` can run one at a time because the output is written to the same file, `sim/vcs/vck_eval.txt`. Where as; the report from `interconnect_analysis` gets directed to `single.log`.

This sample report generated by `$vck_eval` shows the toggle activity. This report can be used directly by the auto partitioner to seed the cost equation or can be analyzed for the purpose of manually developing partition scheme.

Accumulated workload of instances:
 (6179827625)top
 (4164496216)tb.p
 (1905237767)tb.c
 (2489324022)tb.c

The partitioning of the simple computer design is shown in Figure 5. In this case, the partition is done at the first level of the design hierarchy. Auto partitioner can partition the given design in three different modes, normal(+auto_partition), group (+auto_partition_group) and clock (+auto_partition_clock). The normal partitioner will partition instances in the specified module, Group partitioner will allow the user to partition a module or instance into several sub-modules. It will regroup instances in a module and create a new level of hierarchy. Whereas; Clock partitioner will partition the design along clock boundary of storage devices and after partition, all ports on boundary are controlled by clocks. In this example we will use the partitioner in the group mode, +auto_partition_group.

Auto partitioner not only allows the user to automate the design partitioning process for SimCluster distributive simulation, but also implements all the needed dynamic interconnects between the partitions and the SimCluster manager.

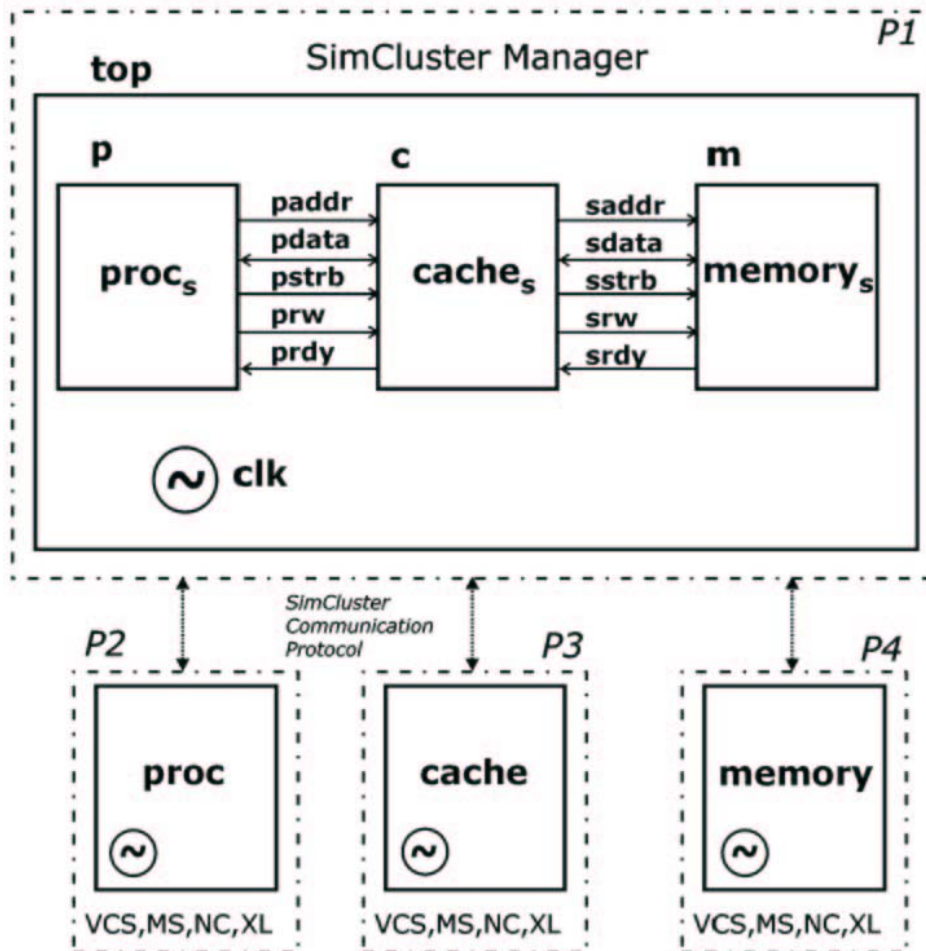


Figure 5. Simulation cluster partitioning of simple computer design

Setup Step 1: The top-level netlist must be in Verilog HDL

Setup Step 2: Input configuration files for +auto_partition_group.

auto_part_run.sh(auto partitioner run script)

```
#!/usr/bin/csh -f
vck.exe +use_mincut +use_mincut +auto_partition_group \
    +design_hierarchy_full \
    -l partitioner.log -f files.f
```

Other configuration files are: `partition.config`, allows the user to set the number of desired partitions, `partition.table`, to specify the seed for the partitioner, and the `clock.table`, has the design clock signal(s) for clock synchronization.

```
partition.config
single_process=2
partition_module = top
partition_number = 3
port_number = 9113
master_host = localhost
partition_mode = 1
rtl_partition=2
```

Above `partition.config` will partition the design into three partitions at the design level top. `single_process=2` option will generate the needed Verilog wrapper for partitions and a `single_sim.v`. This file can be plugged into the single process simulation to verify the correctness of the group partitioner. Sample compile/ simulation script is provided at `sim/vcs/single.sh`. Where as; `rtl_partition` option will partition the RTL/ behavioral code (initial, always, continuous assignments, etc) along with the sub-modules instantiated at the partition level.

```
partition.table
# PARTITON_CACHE 1
top.c
# PARTITON_MEM 1
top.m
# PARTITON_PROC 1
top.p
```

“#” allows the user to provide user specified partition name(s), and “1” at the end of the argument will set the partition to be a hard partition, meaning other than the listed module, no other logic will be moved into the partition. Also note that in the `partition.config`, `partition_number` has been set to three and only three hard partitions have been specified, therefore if we had any other sub-blocks in the top, by default they will be placed into the parent partition.

`partition.table` has another useful option which allows the user to keep logic/ sub-modules in the parent partition. For example:

```
partition.table
%top.c
# PARTITON_MEM 1
top.m
# PARTITON_PROC 1
top.p
```

In this case instance “c” will be kept in the parent partition.

Auto partitioner will output the following files.

`avery_ddmod_master.v` (SimCluster Manager)
`distsim_aux.v` (auto partitioner generated top level netlist with the partition wrapper logic)
`distsim_sync.h` (clock synchronization list for each individual partition(s))
`single_sim.v` (netlist for single process simulation run)
`vchecker.log` (summary of design hierarchy and detailed partition results)
`par_module.txt` (summary of how blocks were allocated)

For example, the `par_module.txt` summarizes the final partition output.

Partitioning Results:

```
top(top):  
  PARTITON_MEM  
  PARTITON_PROC  
  PARTITON_CACHE
```

```
PARTITON_MEM:  
  top.m
```

```
PARTITON_PROC:  
  top.p
```

```
PARTITON_CACHE:  
  top.c
```

Setup Step 3: Auto partitioner adds the SimCluster connect calls to each partition which will enable the simulators to register and attach to SimCluster manager when they are invoked. The connect calls must be made from a module with its timescale set at underlying time precision. This is required based on the specific behavior of the Verilog PLI. Since SimCluster communicates through the `channel_connect` call, this call must be executed from within a module which has its timescale set to the underlying time precision of the entire simulation. In this case the timescale is 1 ns and the time precision is also 1 ns. Often in designs, there are many timescale directives. Special care must be taken to set the timescale of the `channel_connect` to the minimum time precision of all modules.

```
module PARTITON_PROC( prw, pstrb, prdy, pdata, paddr, clk);  
  output prw;  
  output pstrb;  
  input prdy;  
  inout [15:0] pdata;  
  output [7:0] paddr;  
  input clk;  
  
  proc p(clk, paddr, pdata, prw, pstrb, prdy);  
  initial  
  begin  
    $channel_connect("PARTITON_PROC", "localhost", 9113);  
  end  
  
  `include "distsim_sync.h"
```

Setup Step 4: SimCluster needs to have the input and output clock ports on simulation partition boundaries declared. This is done in `avery_ddmod_master.v` using the `channel_clock` command. This

command makes sure clock and data propagation order is correct during simulation of non-blocking assignments in Verilog. Following initial block would have to be added to the `avery_ddmod_master.v`.

```
module avery_ddmod_master;
top_stub top_stub();
PARTITON_PROC PARTITON_PROC();
PARTITON_MEM PARTITON_MEM();
PARTITON_CACHE PARTITON_CACHE();

initial
begin
    $master_channel_total(4, 9113);
    $module_dynamic_link(top_stub);
    $boundary_preserve(top_stub);
    $module_dynamic_link_double(PARTITON_PROC, PARTITON_MEM,
                                PARTITON_CACHE);
    $boundary_preserve(PARTITON_PROC, PARTITON_MEM, PARTITON_CACHE);
    $channel_clock(PARTITON_PROC.clk);
    $channel_clock(PARTITON_MEM.clk);
    $channel_clock(PARTITON_CACHE.clk);
    $socket_traffic_bucket(100);
end
endmodule
```

Setup Step 5: In order to monitor the SimCluster Communication Protocol (SCP) activity, SimCluster provides the `socket_traffic_bucket` facility. This function will monitor SCP reads/writes. This is useful to assess the overhead of the SCP in a parallel simulation run. If it is abnormally high, steps may be taken to optimize the simulation further. Note this command is shown in the example code in Step 4.

Setup Step 6: If your top-level test bench has hierarchical cross-partition references, the partitioner adds the appropriate `channel_variable_map/mirror` directives to map/mirror variable value change with locally duplicated variables. For non-Verilog testbench environments including PLI, SystemC, Vera, or Specman, hierarchical references need to be listed in a `pli.table` file since the partitioner cannot automatically resolve these. In the simple computer, the variable, `dump_memory`, controls whether the cache and memory contents will be dumped at the end of the simulation. Here the code from `cache.v` illustrates how this variable is mirrored in the cache module locally. For more information on cross module references, see Application Note #5, “SimCluster Friendly Testbenches”.

```
reg dump_memory;
integer a;

initial

$channel_variable_mirror("top.PARTITON_PROC.p.dump_memory",
dump_memory);

always @(posedge dump_memory) begin
```

Optimizing Performance

There are several steps that can be taken ahead of time to optimize performance. These include Steps 7-9.

Setup Step 7: Clock based synchronization can be used to minimize the inter-simulator sync times. In numerous benchmarks, using `sync_on` has been shown to reduce SCP messages by a factor of 20X or more. Consider for example gate-level designs (unit delay or full-timing mode), simulation time stepping occurs far more often than at the RTL level. To have good performance with SimCluster, the clock-based

synchronization (sync_on) is highly recommended for mixed level and gate-level designs where pure cycle-based semantics are not the default.

The auto partitioner can perform the setup necessary for clock-based synchronization by the user defining a clock.table file which lists all the clock sources. The partitioner output file distsim_sync.h then will include the \$sync_on directives required. When the design is compiled for parallel simulation, the +define+SYNC_ON compiler directive enabled clock-based synchronization.

Compiling The Simple Computer Design

There are several steps to compile the design. Compile scripts for single-process and SimCluster compilations can be found in the <install_dir>/sim/<simulator> directories which include:

```
vcs - VCS (Verilog only)
nc - NC-Verilog (Verilog only)
```

Each simulation partition is compiled separately. Steps 11-12 are required.

Setup Step 8: If the simulation cluster is an SMP computer, the +vci_unix plusarg should be used which instructs SimCluster to use Unix domain sockets for the SCP. Typically this form of inter-process communication is 2 times faster than TCP/IP.

Setup Step 9: VCS and NCSim require PLI access rights to be properly defined to ensure fast simulation performance. When the models are first compiled in NCSim, use the genaf file options to generate an access file which can be edited to setup minimal required PLI access. Similarly in VCS, the <software_install_dir>tools/tb_release/src/tb_vcs.tab file can be copied and modified to add restricted access for the channel_connect and other SimCluster routines used. The VCS compile script is illustrated below. SimCluster is enabled by setting the +define options, master host and port number. The design runs with the generic PLI tab file with +acc+2 access rights. Finally, the SimCluster library, \$AVERY_HOME/lib/vcs/libtb_vcs.a is linked to the compiled VCS simulator image.

```
!/usr/bin/csh -f

rm mem
rm -r mem.daidir

rm cache
rm -r cache.daidir

rm proc
rm -r proc.daidir

rm parent
rm -rf parent.daidir

vcs \
  -o parent \
  +cli -notice +v2k +acc+2 \
  -P $AVERY_HOME/src/tb_vcs.tab \
  $AVERY_HOME/lib/vcs/libtb_vcs.a \
  ../../auto_part/distsim_aux.v \
  ../../src/verilog/clk_g.v \
  -l parent.log +incdir+../../src/verilog \
  +define+VERBOSE \
  +define+SIMCLUS+AVERY_CHILD_TOP+SYNC_ON +master_port=9113 \
  +plusarg_save +master_host="localhost" +new_inout +vpi
```

```

vcs \
-o mem \
+cli -notice +v2k +acc+2 \
-P $AVERY_HOME/src/tb_vcs.tab \
$AVERY_HOME/lib/vcs/libtb_vcs.a \
../../auto_part/distsim_aux.v \
../../src/verilog/memory.v \
-l mem.log +incdir+../../src/verilog \
+define+VERBOSE \
+define+SIMCLUS+AVERY_CHILD_PARTITON_MEM+SYNC_ON +master_port=9113 \
+plusarg_save +master_host="localhost" +new_inout +vpi
vcs \
-o cache \
+cli -notice +v2k +acc+2 \
-P $AVERY_HOME/src/tb_vcs.tab \
./libtb_vcs.a \
../../../../auto_part/distsim_aux.v \
../../../../src/verilog/cache.v \
../../../../src/verilog/set.v \
-l cache.log \
+incdir+../../src/verilog \
+define+VERBOSE \
+define+SIMCLUS+AVERY_CHILD_PARTITON_CACHE+SYNC_ON \
+master_port=9113 \
+plusarg_save +master_host="localhost" +new_inout +vpi
vcs \
-o proc \
+cli -notice +v2k +acc+2 \
-P $AVERY_HOME/src/tb_vcs.tab \
$AVERY_HOME/lib/vcs/libtb_vcs.a \
../../auto_part/distsim_aux.v \
../../src/verilog/proc.v \
-l proc.log \
+incdir+../../src/verilog \
+define+VERBOSE \
+define+SIMCLUS+AVERY_CHILD_PARTITON_PROC+SYNC_ON \
+master_port=9113 \
+plusarg_save +master_host="localhost" +new_inout +vpi
vck.exe \
../../auto_part/avery_ddmod_master.v \
-l simcluster.log \
+incdir+../../src/verilog \
+define+VERBOSE +define+SIMCLUS+AVERY_CHILD_PARTITON_MEM \
+master_port=9113 +define+HOST="\127.0.0.1\" +new_inout&
vcs -simulate ./parent -l parent.log +new_inout&
vcs -simulate ./mem -l mem.log +new_inout&
vcs -simulate ./cache -l cache.log +new_inout&
vcs -simulate ./proc -l proc.log +new_inout&

```

Running the Simulation

A parallel simulation run is started by invoking SimCluster manager, vck.exe and simulations for each of the partitions. Simulation run scripts and README files can be found in the <install_dir>/sim/<simulator> directories. Run scripts for single-process simulations and SimCluster runs

are provided. The VCS scripts are shown below. When running run_linux.sh on a Linux platform, four xterm windows will pop-up. SimCluster manager will run in one and VCS will run in the other three.

```
dist.sh:
rm mem
rm -r mem.daidir
rm cache
rm -r cache.daidir
rm proc
rm -r proc.daidir
.
.
xterm -e vcs -simulate ./parent -l parent.log&
xterm -e vcs -simulate ./mem -l mem.log&
xterm -e vcs -simulate ./cache -l cache.log&
xterm -e vcs -simulate ./proc -l proc.log&
```

As the simulation starts, SimCluster manager will wait for instances top.p, top.c, and top.m to register. Once each simulation partition initializes and registers, simulation will start. The simple computer design simulation tests writing all memory addresses using several address patterns and then reads the data back. In some cases, the reads will take a cache miss which requires updating the cache from memory, other times the read data will be resident in the cache. The read data is checked when it is returned to the processor. As the simulation runs, the simulation will encounter programmed interrupt points when one of the simulation windows will revert to interactive mode. This illustrates how interactive debug is performed in SimCluster.

Conclusions

The application note presented a step-by-step guide to using SimCluster on a simple computer design including tool overview, how to partition a design, configuration and set-up, and simulation. SimCluster provides a scalable, open, and flexible solution to increase RTL and gate-level simulation performance and capacity by 300-700% or more. The example design illustrates how, with minimal effort, parallel simulation can be utilized in both Verilog and VHDL design methodologies and runs with the most popular HDL simulators.