Parallel VHDL Simulation

Parallel Programming Lab (PPL)
University of Illinois at Urbana-Champaign
Authors: David Kunzman, Dr. Terry Wilmarth, Dr. Laxmikant Kale

Motivation

The complexity of computer components is growing at a substantial rate. Consider modern processors.

- Intel's Pentium 4
  - Prescott Core = 125 million transistors
  - Extreme Edition = 178 million transistors
- AMD's Athlon 64
  - Venice Core = 114 million transistors
  - X2 = 233.2 million transistors

GPUs have even more!

- nVidia 7800 GTX = 302 million transistors
- ATI x1800 Series = 321 million transistors

There is increased market pressure for high performance computing at a low cost.

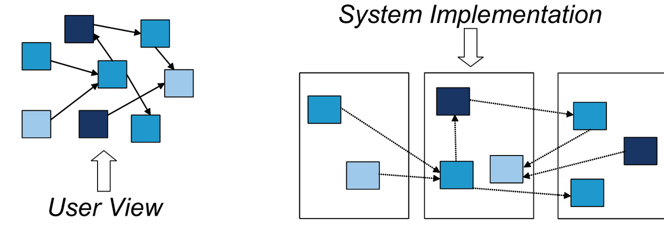
- High Demand for High Performance
  - Scientific Computing
  - Business Servers
  - Gaming Industry (both PC and console)
- Must Stay Competitive to Survive
  - Keep costs down
  - Reduce Time-to-Market / Development-Time
    - Verification of design is becoming a bottleneck in the design cycle

***Natural Solution**: Speed-up simulation through the use of many processors.*

- Problem Sizes are Getting Larger and Larger
  - Requires greater amounts of compute power
  - Requires larger amounts of memory
- Multicore Chips
  - Processor clock frequencies are leveling off
  - To utilize modern processors, applications will need to be designed to utilize multiple processing elements in parallel
    - Multiple chips
    - Multiple cores

Charm++

Charm++ is a runtime system which can be used by parallel programs. In the Charm++ model, the application is broken up into a collection of objects called chares. Each chare is capable of receiving messages from other chares in the application. When the application runs, the chares are spread across all of the processors available to the application.



The mapping of which chare is located on which processor is handled by the Charm++ Runtime System. This relieves the programmer of the details of placing objects manually. The programmer simply sends a message to the target chare and the runtime system takes care of "knowing" where that chare is and making sure the message gets to it's target.

With the addition of "pup routines," which describe how chares can be serialized into messages, the runtime system can automatically load balance the application while the application continues running. This allows the application to better utilize the processors that it is running on. Additionally, the pup routines allow the runtime system to checkpoint an application so its state can be saved. At a later time, an application can be restarted at the exact point in its execution where the checkpoint occurred (in other words, it can pick-up right where it left-off).

For more information, Please Visit the PPL Website: **http://charm.cs.uiuc.edu**

POSE

The Parallel Object-oriented Simulation Environment (POSE) is a framework for performing Parallel Discrete Event Simulations (PDES). In a POSE simulation, each object in the simulation is called a "poser." A poser is a chare with some additional functionality to support PDES simulations.

- What does POSE add?
  - Handles overhead of PDES (programmer would otherwise have to code)
    - Event queues for each poser
    - Keeps track of simulation time
  - Optimistic/speculative execution of events
    - Has several strategies for optimistic/speculative execution of events
      - User defined strategies also supported
    - Periodically checkpoints posers during execution
    - Handles rollback when needed
  - Built on Charm++
    - Works with Charm++ load-balancing framework
    - Works with Charm++ communication library (communication optimization)
    - etc.

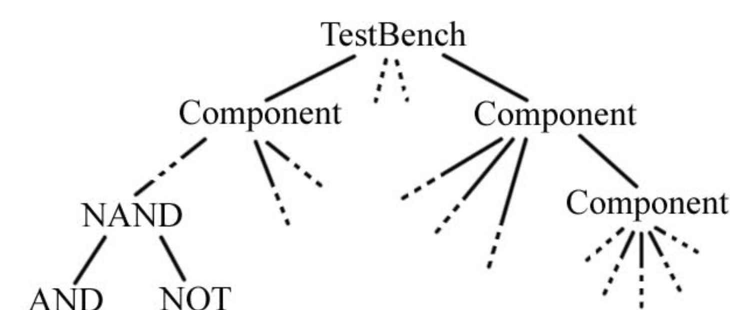
For More Information, Please Visit: **http://charm.cs.uiuc.edu/research/pose/**

VHDL - Basic Overview

VHDL stands for "Very High Speed Integrated Circuit Hardware Description Language". It is a hardware description language (HDL) that is used to model integrated circuits (IC). Another popular HDL is called Verilog.

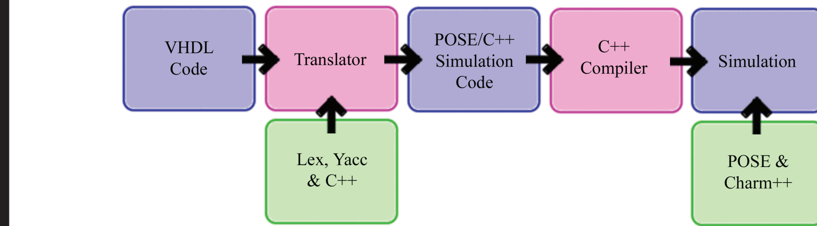
VHDL supports two ways of describing an IC. The first is structural. In a structural model, only simple gate operations (and, or, etc.) are used. More complex "components" are built from simpler "components". The second is behavioral. In a behavioral model, more complex language constructs are used (if statements, loops, etc.) which do not have a direct mapping to hardware.

Each "component" is comprised of two parts. The first is an entity declaration which defines the component's interface to the outside world. The second is an architecture declaration with defines the logic internal to the component. In this poster, the combination of entity and architecture is referred to as a "component". The components can be nested which allows more complex component to be formed from simpler components. This creates a design hierarchy as shown below. (See the "VHDL File" at the top of "Translator Overview" to see code for the NAND sub-tree below).



Translator Overview

The VHDL code is first translated into a series of C++ files. Each entity/architecture pair is translated into its own class. The generated code uses the POSE framework which, in turn, uses the Charm++ Runtime System. Once all of the code has been translated into C++, the simulation can be compiled using a standard C++ compiler.

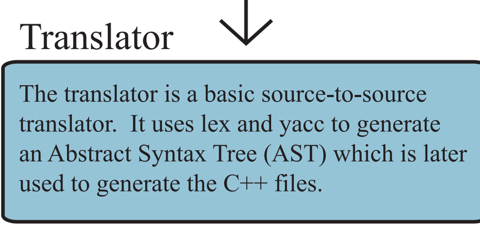


VHDL File

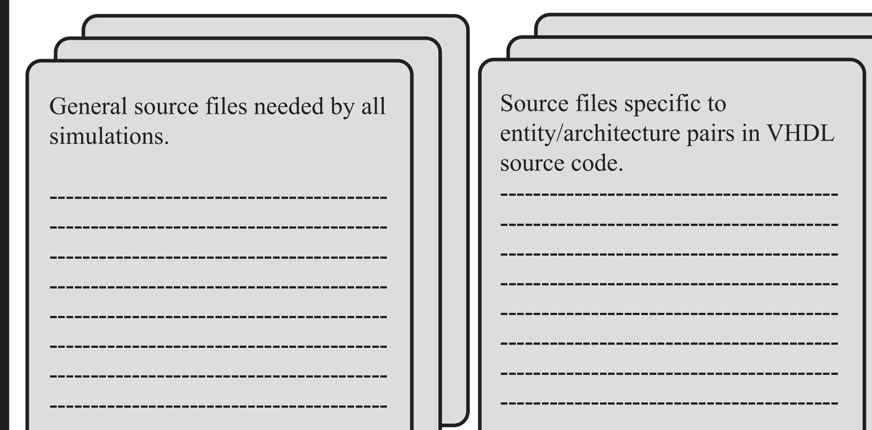
```
entity NOT_GATE is
  port (A: in BIT; Y: out BIT);
end NOT_GATE;

entity AND_GATE is
  port (A, B: in BIT; Y: out BIT);
end AND_GATE;

entity NAND_GATE is
  port (A, B: in BIT; Y: out BIT);
end NAND_GATE;

architecture NAND_GATE_struct of NAND_GATE is
  signal andOutput : BIT;
  component NOT_GATE port (A: in BIT; Y: out BIT); end component;
  component AND_GATE port (A, B: in BIT; Y: out BIT); end component;
begin
  myAndGate : AND_GATE port map (A=>A, B=>B, Y=>andOutput);
  myNotGate : NOT_GATE port map (A=>andOutput, Y=>Y);
end NOT_GATE_struct;
```

Currently, there is only a single special requirement for the input VHDL files. The upper most component in the design hierarchy needs to be named "Testbench". Eventually, as more support for VHDL is added, this requirement should be removed.

Translator

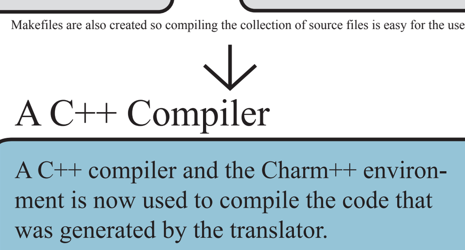


Generated Source Code Files (C++/POSE/Charm++)



Makefiles are also created so compiling the collection of source files is easy for the user.

A C++ Compiler



Compiled Simulation

The output of the C++ Compiler is a single executable that, when executed, will simulate the circuit from the described in VHDL source code. Because both C++ and Charm++ are available on so many platforms, an executable can be created for most machine configurations.

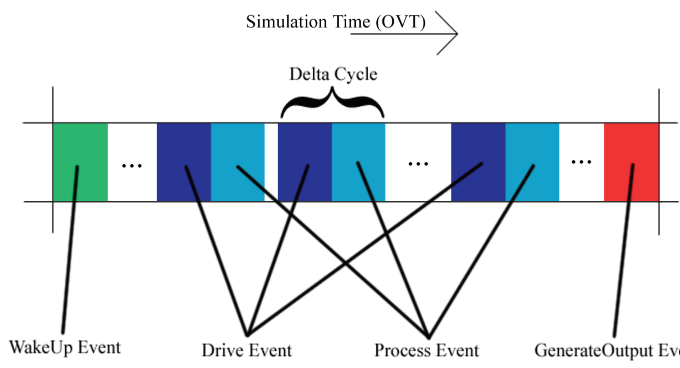
Run the Simulation

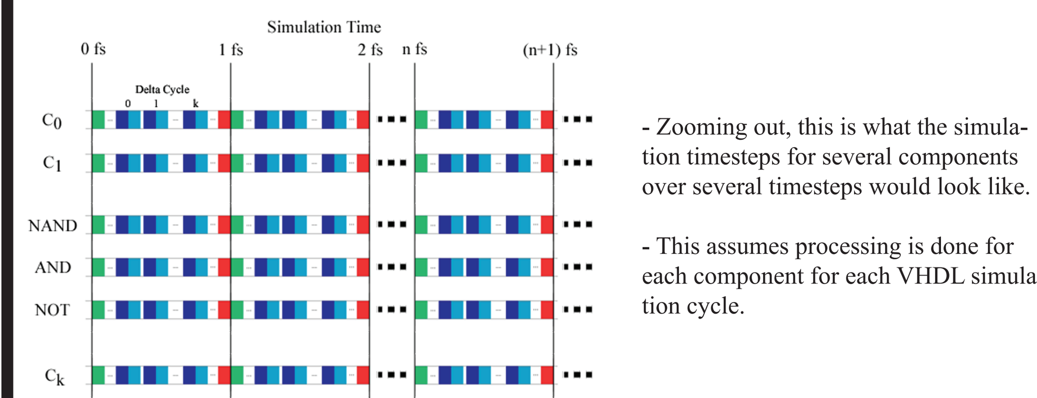
At this point, the compiled simulation is ready to be executed on the target cluster/supercomputer. (See "Simulation Overview")

Simulation Overview

After the VHDL simulation code has been compiled into an executable, this executable can be used to simulate the circuit the VHDL source code describes. Each "instance" of a "component" becomes a separate poser. Messages are used to pass the values of ports between components.

Each VHDL simulation cycle is represented as zero or more drive events followed by a single process event. The VHDL simulation cycles are grouped so that each grouping starts with a full VHDL simulation cycle followed by all the VHDL delta cycles that occur before the next full VHDL simulation cycle. In other words, simulation time only changes at group boundaries.



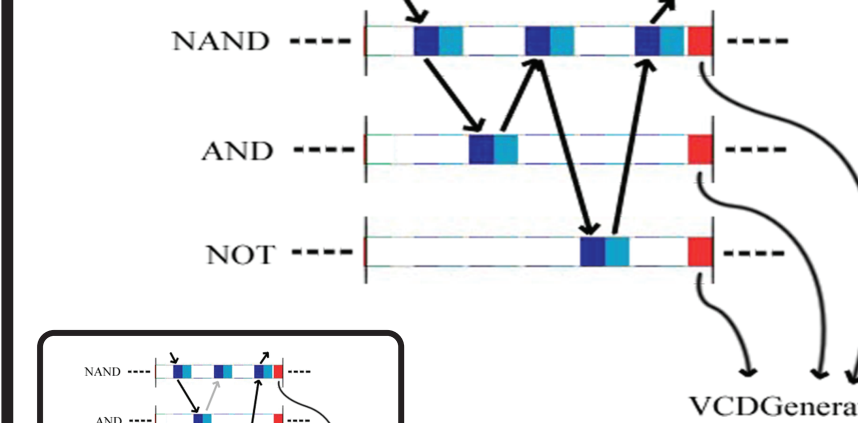


- Zooming out, this is what the simulation timesteps for several components over several timesteps would look like.
- This assumes processing is done for each component for each VHDL simulation cycle.

- Not all components do work each cycle
  - Timelines look more like this
  - Components skip over empty cycles
- A component only does work on demand (when it receives a message)
  - Send self a wakeUp
  - Send self a drive (earlier in time)
  - Other component sends a drive

Simulation Time
0 fs 1 fs 2 fs n fs (n+1) fs
Delta Cycle
NAND
AND
NOT

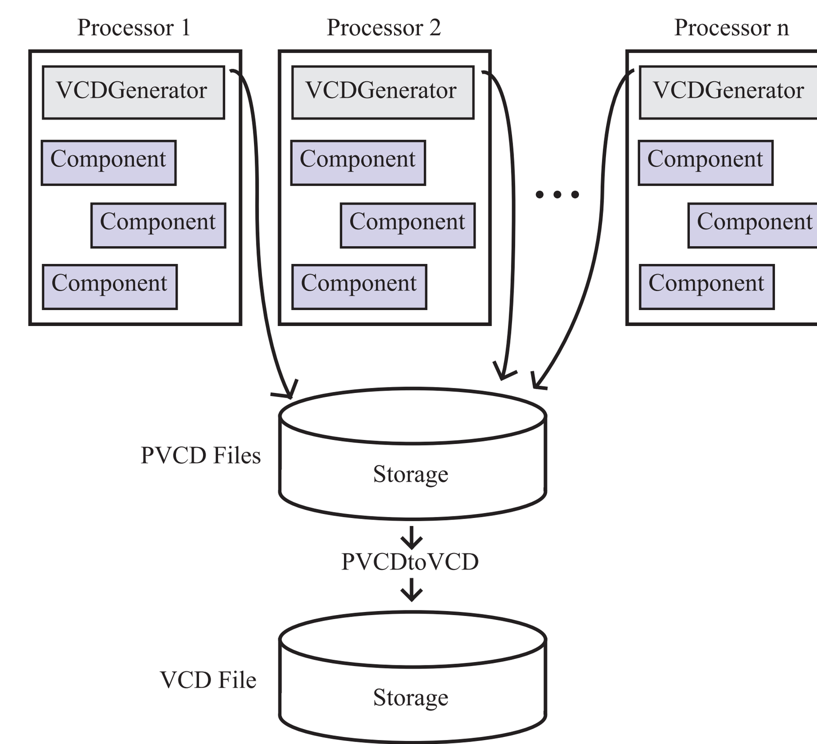
As components update their values (ports, signals, etc.), any values that need to be passed onto other components are passed through messages. The figure below illustrates what would happen if a drive message for a input to a NAND_GATE (see VHDL File example) was received. (Downward arrows relate to inputs and upward arrows relate to outputs)



NAND
AND
NOT
VCDGenerator

As a later optimization, the translator can remove messages from the critical path if it determines the receiving component just forwards the value on.

As values (ports, signals, etc.) are modified, the values changes are passed onto special posers called VCDGenerators. Each processor has a single VCDGenerator. VCDGenerators are responsible for recording all value changes that occur on that processor to disk. A single pvcd file is generated per processor. These files are later combined to generate a vcd file which can be opened with a vcd viewer of the user's choosing.



VHDL Support So Far

The translator supports a subset of the VHDL language. Arbitrary VHDL files can be parsed as long as they only use the supported subset of the VHDL language. Currently, there is only one special requirement for the VHDL file; the root component in the design hierarchy must be named "Testbench". Though, this requirement should be removed with the addition of an additional IR (see "Future Work").

Some Specific Items Include:

- Most Expression Operations
- Several Basic Types Defined in VHDL
- Concurrent Signal Assignments
- Concurrent Processes
- Most Sequential Statements
- etc.

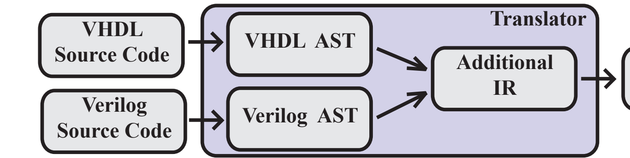
Some specific circuits that have been designed and simulated include:

- Clock generator (continuous processes)
- Mux (test simple control signals)
- Counter (test simple control signals, some behavioral constructs)
- Parity Generator (several sub-components that interact, test correctness of message propagation)
- etc.

Future Work

Translator:

- More support needs to be added for the VHDL language. Currently, the translator only recognizes a small subset of the VHDL language.
- Support for other HDL languages (specifically Verilog) and mixed-HDL simulations.
- Create an additional intermediate representation (IR) that can be used as a storage format for compiled components and structured such that optimizations/transformations are easy to apply
  - Flattening of the design hierarchy
  - Splitting/combining components for better granularity
  - Removing messages from the critical path



Simulation:

- Create strategies specifically designed to accommodate the behavior of components and VCDGenerator posers (specifically, the difficulty of 'clock' based simulations).
- Further development of load balancers for the POSE framework.
- With additional support for the VHDL language, comparison of this simulator to other VHDL simulators using some standard benchmarks.

Initial Results

VHDL Design File Used

- Clock Generator
- 12bit Counter (counts clock pulses)
- Parity Generator
  - Generates odd parity for output of counter
  - Tree of XOR gates and single NOT gate

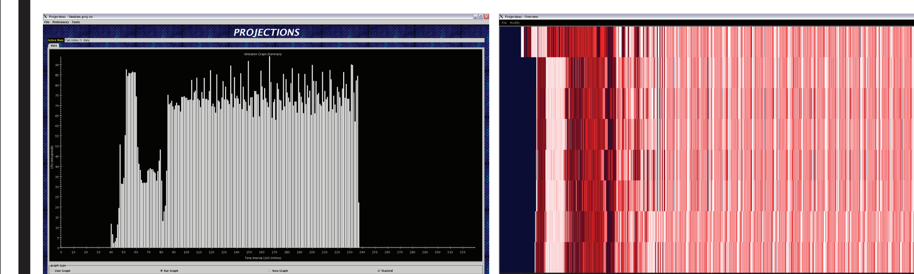
Note: For these tests, the parity generator was duplicated many times to increase the number of components in the simulation.

Machine used for Testing

- Turing Cluster at UIUC (see http://www.cse.uiuc.edu/turing for details)

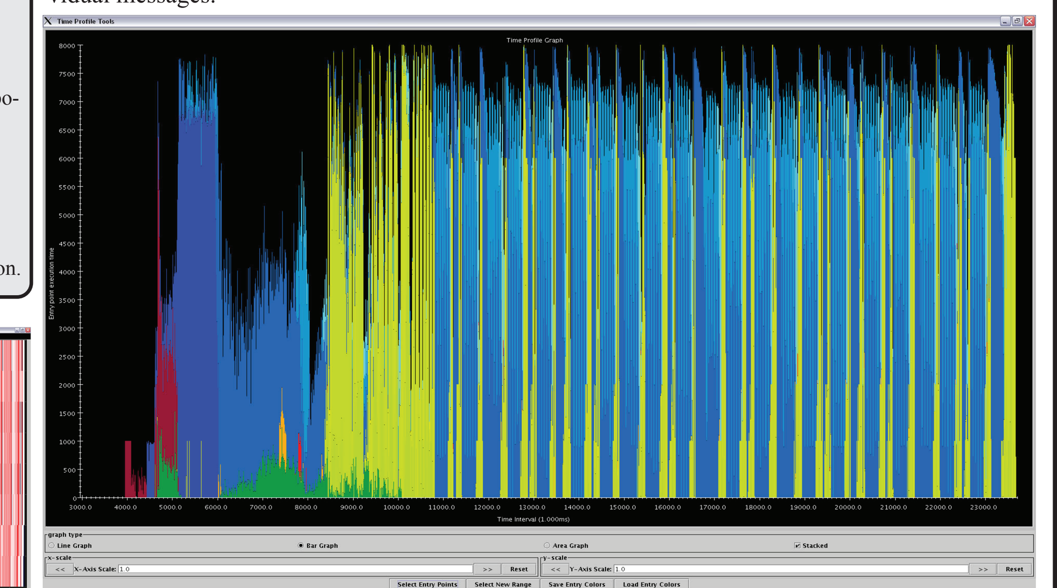
Note: The projections images below represent a runs that was made without collecting vcd information.

The upper three graphs in this section are from a performance visualization tool called Projections. Projections has been developed along side of Charm++ and is a powerful tool for allowing a developer to understand the behavior of an application. With the profile data it collects, it can display a wide variety of information from the high-level graphs shown here to the low-level details of individual messages.

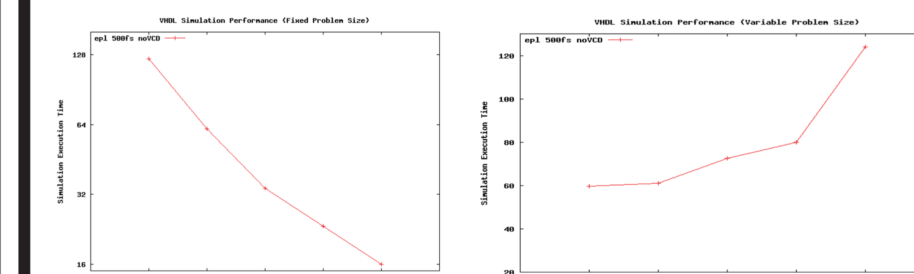


This is a Projections summary graph. It shows the overall utilization of all the processors as the simulation runs. As can be see from this graph, the utilization of the processors is approximately 70% to 75%. Clearly, more work is needed to speed up both sequential portions of the simulation along with increasing parallelism so the processors are better utilized overall.

This is a Projections overview graph. It shows the utilization of each individual processor as time passes. As can be seen from this image, the overall load balance between the processors is fairly even. The white indicates 100% utilization, black indicates 0%, and shades of red are used for the percentages in-between (the lighter the shade of red the higher the utilization).

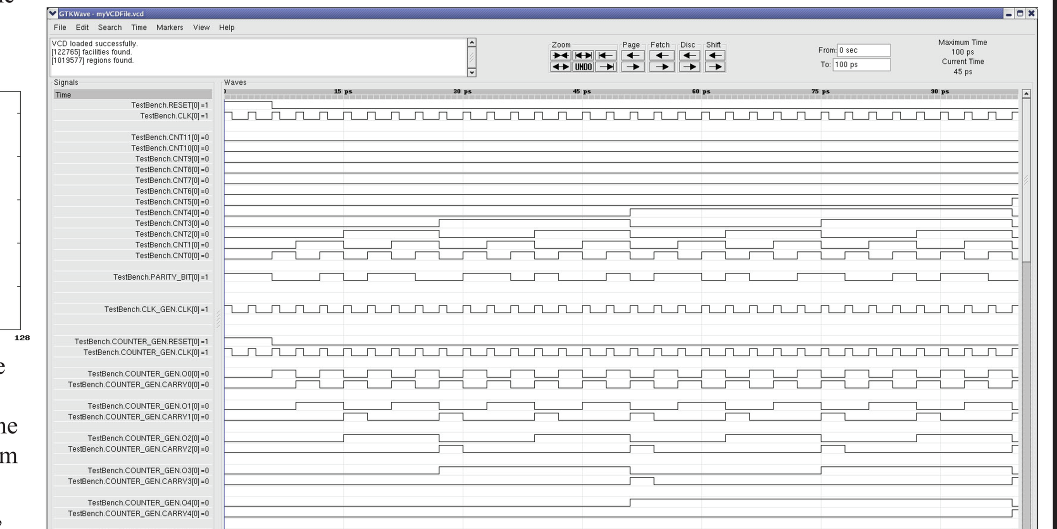


This is a Projections time profile graph. It shows the amount of time spend executing the different entry methods (message handlers) during the course of the simulation. The shades of blue relate to the various entry methods of the components (wakeUp, drive, process, generateOutput). The other colors relate to various other entry methods used in both the POSE framework and the Charm++ runtime system. In the beginning of the simulation (left) there is a great deal of overhead (initial values being passed around, etc.). Once, the simulation gets passed this initial overhead, it stabilizes into a more clear routine with less overhead (in particular, the narrow yellow spikes in the right half of the graph are GVT calculations that POSE uses to move the simulation time forward.)



This graph shows the performance of the simulator as more processors are used to simulate the exact same problem (as the number of processors doubles, the work per processor halves). As can be seen from the graph, scaling works fairly well up to 32 processors. After 32 processors, doubling the processor count no longer halves the execution time.

This graph shows the performance of the simulator as the problem size is varied with the number of processors (double the number of processors, double the problem size; a horizontal line represents perfect scaling). As can be seen from the graph, scaling is alright up to 32 processors. Both performance graphs indicate there is still plenty of scaling work to be done.



The above picture is a screenshot of the gtkwave waveform viewer being used to inspect signal values generated by the simulation. Gtkwave can be found at: http://www.cs.manchester.ac.uk/apt/

Another POSE Related Project: BigNetSim

Architecture and Objective of BigNetSim

Performance visualization (Projections)
Offline PDES
Network Simulator
BigNetSim (POSE)
Simulation output trace logs
Charm++ Runtime
Online PDES engine
Load Balancing Module
Performance counters
Instruction Sim (RSim, IBM, ..)
Simple Network Model
BigSim Emulator
Charm++ and MPI applications

Motivation:

- How to write a Peta-Scale parallel application?
- What will be the performance like?
- Will the applications scale?

Objective:

- Develop techniques to facilitate the development of efficient peta-scale applications
- Based on performance prediction of applications on large simulated parallel machines

Simulation-based Performance Prediction model:

- Focus on Charm and AMPI programming models – Structured Dagger
- Performance prediction based on PDES

Simulation Details

Target Node
Target Node
Converse scheduler
Real Processor
Bigsim Emulator – Functional Diagram

- Emulate large parallel machines on smaller existing parallel machines – run a program with multi-million way parallelism
  - Ensure Time-stamp correction
- Emulator layer API built on top of machine layer
- Charm++ implemented on top of emulator
- Emulator layer supports all Charm++ features:
  - Load-balancing
  - Communication opt

Predict Performance based on:

- Processor Model – wallclock time; user supplied timing info.
- Network Model – contention based Interconnection Networks

Network Simulation Details:

- Can specify any network for the machine being simulated
- Implements Collective hardware communication

BGproc
Task
BGnode
Message
Channel
Switch
Net Interface
Packet
Config
Machine Params
Input VC Selection
Output VC Selection
Position
MsgStore
Topology
Routing Algorithm
Header
Routing Information
Flowstart
Remote Message ID

Interconnection Network Simulator in BignetSim

For More Information, Please Visit the PPL Website: http://charm.cs.uiuc.edu

Created 11/30/05