Proceedings of the 2005 Winter Simulation Conference
M. E. Kuhl, N. M. Steiger, F. B. Armstrong, and J. A. Joines, eds.

PARALLEL SIMULATION OF PETRI NETS ON DESKTOP PC HARDWARE

Robert Geist
Jacob Hicks
Mark Smotherman
James Westall

Department of Computer Science
Clemson University
Clemson, SC 29634-0974, U.S.A.

ABSTRACT

A comparatively simple approach to highly parallel simulation of Petri nets on commodity, desktop PC hardware is suggested. A mapping, described in the programming language *Cg*, of Petri net semantics to the SIMD architecture of NVidia 5-series and 6-series GPUs is provided, and a prototype simulator is tested on both conflict-intensive and conflict-free Petri net models. In all cases, the prototype parallel simulator is seen to deliver substantial performance gains over its serial counterparts. Limitations of the approach and open design issues are also described.

1 INTRODUCTION

Among well-known tools for modeling computer and communication systems, Petri nets continue to play a prominent role of significant importance. Recall that a Petri net is a directed bipartite graph whose two vertex sets are called *places* and *transitions*. Places are traditionally represented by circles and transitions by rectangles. Places may contain one or more *tokens*, represented by small discs. The semantics attached to such nets are rules for simulation:

- If every input place to a transition contains one or more tokens, the transition is *enabled*.
- Enabled transitions may *fire*, that is, remove one token from each input place and add one token to each output place.
- If firing an enabled transition would disable a concurrently enabled transition (conflict), the firing transition is chosen at random.

A common extension to the basic Petri nets, *inhibitor arcs*, adds Turing completeness: If an arc from a place to a transition is an inhibitor arc, the transition is enabled only if the place is empty.

Due to the ease with which modelers can represent common system features such as concurrency and resource contention, Petri nets and their extensions have been used by many authors in modeling both the reliability and the performance of computer and communication systems. The most common extension is probably the Generalized Stochastic Petri Net (GSPN) (Marsan, Conte, and Balbo 1984) in which transitions may fire instantaneously (when enabled) or have exponentially distributed delay between enabling and firing. Such nets can be transformed into discrete-state, continuous-time Markov processes so that analytic solution techniques may be employed to extract both steady-state and transient information.

Nevertheless, in spite of significant advances by numerous researchers in obtaining analytic solutions of timed Petri nets (Marsan and Chiola 1987; Choi et al. 1993a; Choi, Kulkarni, and Trivedi 1993b; German and Lindemann 1993), simulation remains the only viable alternative for many classes of nets, particularly those having large numbers of transitions with non-exponential firing times. Thus interest in accelerated simulation through parallel processing remains strong. Several authors have investigated parallel and distributed Petri net simulations. Nicol and Roy (1991) considered distributed execution in the framework of communicating discrete event simulations. They offered both a communication protocol and an event priority scheme that allowed significant speedups over sequential simulation. Thomas and Zahorjan (1991) also achieved significant performance improvements over sequential simulation by modifying the semantics of Chandy and Misra's classical model for parallel simulation (Chandy and Misra 1981) to allow for Petri net semantics. In particular, they proposed a *selective receive* mechanism whereby a logical process may ignore some of its input channels in computing its own message acceptance horizon. Ferscha (1994) provided an excellent distillation of the issues involved in both parallel (SIMD) and distributed simulation of Petri nets. This

included a description of the innovative technique, due to Baccelli and Canales (1993), for parallel simulation of certain classes of nets using matrix recurrence equations.

The purpose of this paper is to suggest a comparatively simple alternative to these approaches that is based on an ongoing "revolution" in PC-class, desktop hardware. Graphics processing units (GPUs) have become increasingly attractive targets for general purpose (non-graphics) computation. Designed originally to speed the rendering of images to PC displays, GPUs are now fully programmable and, largely due to their highly-parallel, SIMD designs, significantly outperform even high-end CPUs on common tasks. For example, on peak performance measures, a 3 GHz Intel Pentium 4 CPU is rated at 12 GFLOPS and 6 GB/sec memory bandwidth; the 400 MHz NVidia 6800 GPU is rated at 45 GFLOPS and 36 GB/sec memory bandwidth (Hanrahan 2004). Nvidia now provides both a high-level programming language for GPUs, *Cg*, and the software tools necessary to integrate *Cg* programs into ordinary *C* programs that use a standard graphics API such as *OpenGL* or *DirectX* (Fernando and Kilgard 2003). Our goal here is to provide a *Cg*-based mapping of Petri net semantics to the SIMD architecture of the NVidia GPUs and thereby offer a low-cost, high-performance technique for Petri net simulation on the desktop PC. As of this writing, a graphics card with a 6800 Ultra GPU, as tested here, can be purchased for approximately $380.

It is worth noting that numerous, non-graphics applications have already been implemented on GPUs, and thus a body of technique, upon which we have drawn, is beginning to emerge. These applications include the fast Fourier transform (FFT) (Moreland and Angel 2003), sparse linear equation solvers (Bolz et al. 2003), and multi-grid solvers for boundary value problems (Goodnight et al. 2003).

The remainder of the paper is organized as follows. In the next section we discuss *Cg*, the Nvidia 6800 architecture, and vertex and fragment programming. In Sect. 3 we provide a simulator design and discuss the choices made in allocating tasks between the GPU and the CPU. In Sect. 4 we compare the performance of our simulator against two sequential simulators, *xpetri* (Geist et al. 1994), and *SPNP* (Hirel, Tuffin, and Trivedi 2000), on two Petri net models, each of which can be of arbitrary size. In Sect. 5 we offer a discussion of the current limitations of our approach, and, in Sect. 6, we suggest conclusions and some open questions.

2 CG AND THE NVIDIA ARCHITECTURE

2.1 The Architecture

A GPU is designed to implement the traditional graphics pipeline. Surfaces to be rendered are specified as collections of triangles, and these triangles are represented by triples of vertices in $\mathbb{R}^3$. The graphics pipeline will first transform the vertices, an operation which includes application of affine transformations to alter their positions and application of a lighting model to compute their colors. The pipeline will then assemble triples of vertices into triangles and *rasterize* the transformed triangles into fragments, which are "pre-pixels". Rasterization includes a determination of which pixels will be covered by each triangle and a linear interpolation of per-vertex information across those pixels to yield per-fragment information. The fragments are then processed one-by-one, often using *textures* to provide additional surface detail and additional lighting effects. Textures are 1, 2, or 3-dimensional images (often 2D digital photographs), and texture elements typically contain three or four color values, e.g., RGBA. The final results are pixels, which are written into the frame buffer and displayed.

Fragments are isolated from one another and are processed independently. A texture memory, or cache, which is on-board the graphics card, can be accessed in a read-only manner during fragment processing to obtain texture values. Once the processing is complete for a given fragment, the resulting pixel is written into the frame buffer. A feedback loop is possible since the contents of the frame buffer can be written out as a texture. This loop is relatively fast because the information copied does not leave the graphics card.

An overview diagram of the graphics pipeline in a GPU is shown in Figure 1.

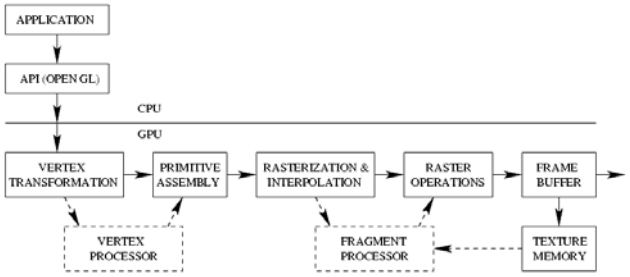

Figure 1: Graphics Pipeline with New Processing Units

The recent, revolutionary advance in GPU architecture is the inclusion of user-programmable processing units, shown in dotted lines in Figure 1. These processing units, one for vertex transformation and one for fragment transformation, allow the user to intercept pipeline values, alter them at will, and insert them back into the pipeline. These units (sometimes called shaders) have become increasingly rich in capability and now support high-level languages such as *Cg*.

Compared to a CPU, a GPU typically has a relatively large number of ALUs and a relatively small instruction memory. The multiple ALUs provide for an enormous amount of data parallelism. Each ALU typically supports four-wide, single-precision floating-point vector operations. When considered for general-purpose computation, the frag-

ment processing units (FPUs) are often more attractive than the vertex processing units because there are typically more of them (On the Nvidia 6800 there are 6 vertex processing units and 16 fragment processing units.) and because arbitrary texture elements can be directly read and indirectly written by the fragment processing units.

A generalized diagram of a fragment processing unit is shown in Figure 2 (Buck and Hanrahan 2003). In the

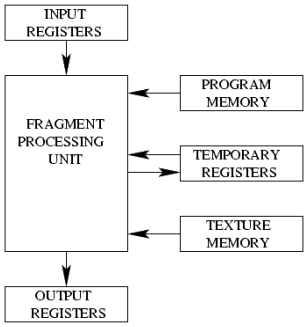

Figure 2: Fragment Processing Unit

standard FPU execution cycle, the input registers to each unit are loaded with up to 16 single-precision floating-point values that represent a fragment, and the temporary registers are initialized to zero. The fragment program is then executed on the data in the input registers, with read-only access to textures (i.e., lookups) and read/write access to the set of temporary registers. The results of processing the fragment, typically a 32-bit color, are written into the output registers.

This approach to individually processing fragments provides a large number of independent computations that cannot affect each other. Because of this independence, a GPU is thus a highly-capable SIMD processing design that can hide memory latency and fully utilize multiple ALUs.

2.2 Cg

Cg (C for graphics) is a high-level language that may be used to write both vertex and fragment programs. It was developed by Nvidia for their graphics cards, but it is now supported on other cards (e.g., those from ATI Technologies) as well. It is freely available, along with extensive documentation, from the Nvidia website, <www.nvidia.com>. It installs easily, and *Cg* programs are easily integrated with ordinary C programs that invoke either the OpenGL or Direct3D graphics API. Multiple vertex and fragment programs may be compiled and downloaded to the graphics card, but only one of each may be active at any instant.

The syntax of *Cg* is C-like, but there are important restrictions. Pointers are not available, nor are some of the standard control flow mechanisms. Until very recently, conditional execution of code was always implemented with predicated instructions (no true branching), and loops were always unrolled by the compiler. Further, there was an execution limit of 1024 instructions. For the most recent compiler (1.3) on the Ultra architecture, some of these limits have been removed (e.g., the instruction limit is 65,536, and true branching is available at the machine level), but common control mechanisms such as **switch, break, continue, goto** and early **return** are not yet available at the *Cg* level.

In spite of these limitations, *Cg* provides many useful extensions to C, including vector types, e.g. **float4**, matrix types, e.g. **float4x4**, and overloaded operators that permit vector addition, and vector-scalar multiplication. Examples, in the context of a Petri net simulator design, will be presented in the next section.

3 A CG PETRI NET SIMULATOR DESIGN

We focus first on the simulation of classical, non-timed Petri nets without inhibitor arcs, arc multiplicities, colored tokens, or any of the myriad of extensions that have been proposed. The approach we have taken does not preclude adding these extensions, but they will obscure the design, and so we defer them.

3.1 Design Overview

As noted earlier, there is a feedback loop available between the frame buffer and on-board texture memory. As a result, textures may be regarded as data arrays available for general purpose computation. The data is stored (encoded) as color values in the texture elements.

Since the CPU and GPU operate asynchronously, an application must first bind the fragment program and the necessary textures to the GPU memory through calls to a graphics API, in our case OpenGL. A call from the OpenGL level to draw any region of the screen then initiates a rendering pass. The fragments targeted in this rendering pass will all execute, in parallel, the bound fragment program and update the frame buffer with their outputs. The application can then copy the frame buffer contents back to a texture and repeat these actions.

In such computations, it is often the case that separate steps must be implemented as separate fragment programs, each of which uses texture reads and arithmetic operations. As outlined above, these fragment programs can write only a single output pixel at a time, in isolation from all the other processing being performed. Thus, a complete update to the data may require a series of fragment programs that must be individually bound in sequence, with each requiring a separate rendering pass from the OpenGL level to invoke the bound fragment program.

```
for(loop=0;loop<MAXLOOPS;loop++){
    write_transitions();
    write_reservations();
    resolve_conflicts();
    write_places();
    }
```

Figure 3: C/OpenGL Side Control Loop

3.2 Design Detail

We use six textures and three fragment programs. Three of the textures are static and are simply loaded into texture memory during program initialization. These textures contain (with some redundancy) place-transition arc information as follows:

- **tex_tip[transitions][max_degree_input]** holds, for each transition (row), a list of indices that identify its input places.
- **tex_pit[places][max_degree_input]** holds, for each place (row), a list of indices that identify its input transitions.
- **tex_pot[places][max_degree_output]** holds, for each place (row), a list of indices that identify its output transitions.

Note that each of these is a 2D texture.

The remaining three textures are dynamic. We use one, **tex_p[PLACES]**, to hold the token counts of the places, another, **tex_t[TRANSITIONS]**, to hold the enabled/disabled states of the transitions, and the third, **tex_r[PLACES]** to count token reservations, i.e., per place, the total number of enabled output transitions that would fire if sufficient tokens were available. All three textures are 1D, so we may update any one by simply drawing a line of pixels from the C/OpenGL side of the program.

Our central control loop, on the C/OpenGL side of the program, then takes the form shown in Figure 3. Each of the calls to *write_transitions()*, *write_reservations()*, and *write_places()* binds a *Cg* fragment program and then draws a line of pixels to invoke fragment processing. Each then uses the OpenGL call, *glCopyTexSubImage1D()* to copy the updated frame buffer contents back to texture memory.

The fragment program invoked by the call, *write_transitions()*, is shown in Figure 4. Several items are worth noting. Parameters have qualifiers *in*, *out*, and *uniform*. The *in* parameters are standard, call-by-value. The *out* parameters are call-by-result or copy-out. These have no counterpart in C, where passing an address is required. The *uniform* parameters are those whose values are set by the C/OpenGL side of the program. They are used here to pass two textures, the places texture, **tex_p**, and the static **tex_tip** texture described earlier. Note that all fragments receive the same values for the uniform parameters on each iteration. The trailing qualifiers, *:TEXCOORD0* and *:COLOR*, indicate pipeline values. In this case, the pipeline provides an input texture coordinate, which serves as an identifying index for the target transition, and the pipeline receives a color which encodes whether or not the transition is now enabled. The algorithm encoded here is straightforward. We use the pipeline-supplied transition index to look up its list of input places. For each of those we look up the current token count. If the product of the token counts is greater than zero, we enable the transition by writing a 1 in the pipeline output color. The *Cg* function **sign** returns 1 if the argument is positive, 0 if the argument is 0.0. There are two reasons we maintain a product rather than testing token counts inside the loop. First, conditionals in *Cg* are relatively slow operations and should be used sparingly. Second, even if we find an empty place early in the loop, we cannot exit from the loop, since neither **break** nor **goto** nor **return** is yet available.

Indexing into textures is somewhat unusual. Indices are floating point values that represent pixel locations. Integral values represent pixel boundaries, and thus we use offsets of 0.5 to sample pixel centers.

Pixels values (output color) are also floats. Visible framebuffers are typically organized with 32-bits per pixel value. These are usually arranged as 4, 8-bit integers that control red, green, blue, and alpha values. (Alpha is for blending.) Nevertheless, most OpenGL implementations allow read/write access to off-screen buffers on the graphics card, and these off-screen buffers (called *pbuffers*) can be organized as 32-bit floats. Using this organization allows us to avoid awkward encodings that would be required for Petri nets with more than 256 places, 256 transitions or 256 tokens in a single place.

The fragment programs invoked by the calls to *write_places()* and *write_reservations()* are similar to this one in both size and syntax, and so they have been omitted here. (The complete simulator source code may be obtained from the website <www.cs.clemson.edu/~rmg/cgpetri.html>.) In the fragment program invoked by *write_reservations()*, each fragment (one per place) independently counts the number enabled among the output transitions for that place and stores this value in the texture **tex_r**. Since the place's token count is stored in **tex_p**, **tex_r[i]**>**tex_p[i]** indicates that place **i** is over-subscribed, and conflict resolution must (later) be invoked. In the fragment program invoked by *write_places()*, each place fragment increments its token count by the difference, (#enabled input transitions - #enabled output transitions), which captures the effect of firing all (enabled) transitions simultaneously.

In our initial design, conflict resolution was also performed on the card using standard voting techniques. In one pass, each transition fragment can write a random value,

```
void enable(in float2 transition_index:TEXCOORD0, out float color:COLOR,
                uniform samplerRECT tex_tip, uniform samplerRECT tex_p)
{
float2 incidence_index, place_index;
float tip_col, incidence_value, place_value, enabled_product;

enabled_product=1.0;
for(tip_col = 0.5; tip_col < MAX_T_IN_DEGREE; tip_col += 1.0){
                incidence_index = float2(tip_col, transition_index.x);
                incidence_value = texRECT(tex_tip, incidence_index);
                place_index = float2(incidence_value+0.5, 0.5);
                place_value = texRECT(tex_p, place_index);
                enabled_product *= place_value;
                }
color=sign(enabled_product);
}
```

Figure 4: Fragment Program Invoked by *write_transitions()*

and, in a second pass, each can determine a winner. Unfortunately, this proved to be a performance bottleneck. As observed earlier, conditionals on the card are expensive, and our conflict resolution algorithm requires numerous instances. It is an aggressive algorithm which selects at random from among all legal enabling choices that provide a maximal number of firings.

We moved conflict resolution back to the C/OpenGL side, and performance improved markedly, even though additional overhead was incurred. Place, transition, and reservation pixels must be read back (to main memory) from the frame buffer, and transition textures must be reloaded (from main memory) after conflict resolution. It is entirely conceivable that an alternative design for conflict resolution would allow this step to remain on the card and thereby offer performance improvements.

4 RESULTS

We implemented this design in a simulator, *cgpetri*, that we tested on two parameterized Petri net models, *dining philosophers*, and *lattice-Boltzmann flow* described below. Our test platform was a dual-processor PC equipped with 2.0GHz AMD Opteron processors, 2GB main memory, and an Nvidia 6800 Ultra graphics card with 256MB memory. Graphics cards with 512MB memory are now available, and thus memory constraints on problem size are comparable to those imposed by standard PC main memory. Our API was OpenGL 1.5, and our operating system was Linux 2.6.9. For each net model, we compared *cgpetri* with two sequential simulators, *xpetri* (Geist et al. 1994), and the well-known, *Stochastic Petri Nets Package (SPNP)* (Hirel, Tuffin, and Trivedi 2000). Originally designed as an analytic modeling tool, *SPNP* now has full simulation capabilities.

4.1 Dining Philosophers

The *dining philosophers*, due to E. Dijkstra, is a classic problem in the synchronization of concurrent processes. Each of *N* philosophers, seated at a circular table, attempts to alternate between"thinking" and "eating". There is one fork placed between each pair of philosophers, but conflict arises because it requires two forks to eat. A (segment of a) Petri net model of the problem is shown in Figure 5. Simulation of this net obviously requires a significant

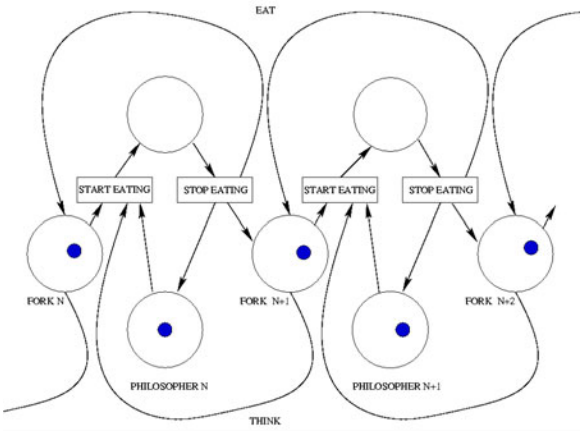

Figure 5: Segment of the *dining philosophers* Net

amount of conflict resolution, and so we might expect, a priori, that the serial simulators would rival the parallel one in overall performance. In Figure 6 we show the results of executing this model on each of the simulators for 100,000 firing epochs. Each data point represents the mean of 30 trials. We computed 95% confidence intervals about each mean value shown, but the largest interval found was 0.258549, and so they have been omitted from the figure.

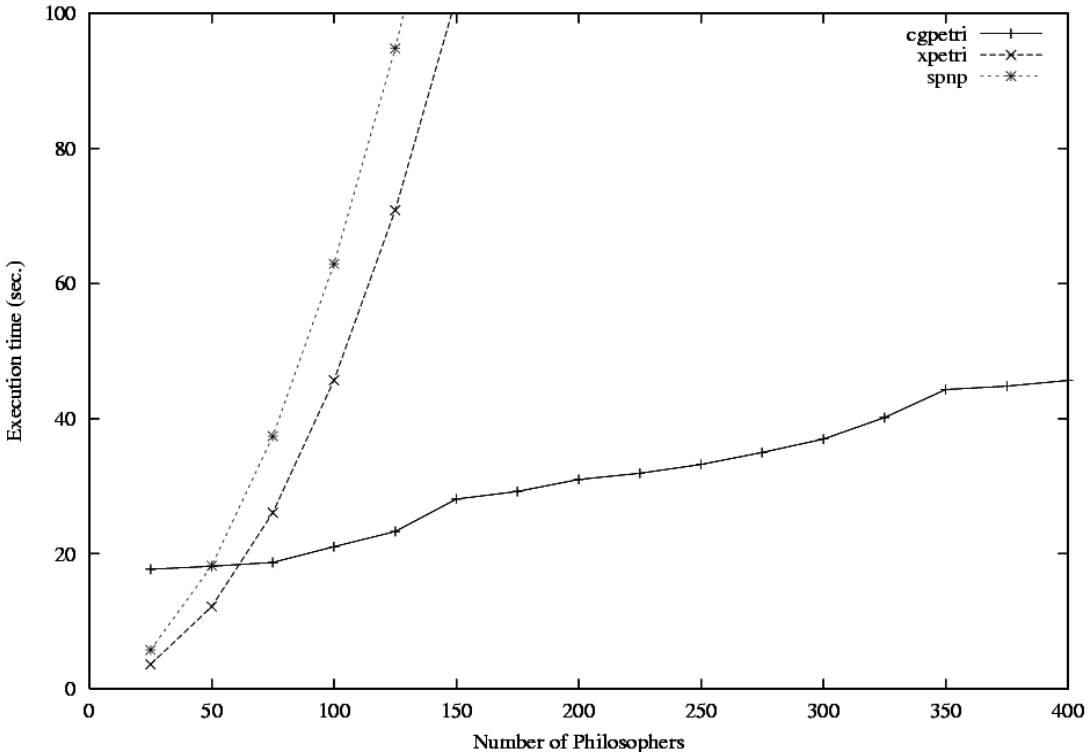

Figure 6: Dining Philosophers Execution Time

Somewhat contrary to expectations, the parallel simulator wins handily, once the net is sufficiently large to overcome an initial overhead cost associated with the parallel implementation. The execution time of *cgpetri* is $O(max\{|places| \times MTD, |transitions| \times MPD\})$ where MTD and MPD denote the maximum degree of the transition nodes and maximum input degree of the place nodes in the graph. Thus, execution time that is approximately linear as a function of the number of philosophers is not surprising. The staging effect in the graph may be attributed to the internal load balancing across the 16 available ALUs. Most serial Petri net simulators are $O(|places| \times |transitions|)$, and thus the slightly super-linear performance for *SPNP* and *xpetri* on the nets studied is not surprising.

4.2 Lattice-Boltzmann Flow

Lattice-Boltzmann methods offer a computational alternative to finite-element methods for solving multi-dimensional systems of coupled PDE's. They are particularly well-suited for modeling generalized fluid transport problems in two or three dimensional Euclidean space. In a lattice-Boltzmann model, a regular lattice is embedded in the problem space. The model is parameterized by a set of directions in which generalized fluid densities travel and a collision matrix which defines how incoming densities are dispersed at each lattice point for each time step. The accuracy of a Lattice-Boltzmann model improves as the density of lattice points increases. Therefore, especially in the three-dimensional case, both CPU and memory requirements can become so large as to mandate a parallel solution. A recent application to photon transport in diffuse media in which the solution was distributed across 64 nodes of a Beowulf-type cluster is described in (Geist et al. 2004).

In the parallel solution, the global lattice is decomposed into subcubes of identical size, and each subcube is assigned to a dedicated computing node. Even though the subcube partitioning minimizes communication overhead, a significant amount of communication overhead remains. Before each iteration of the lattice-Boltzmann update, each node must export a subset of the directional densities for each lattice point that is adjacent to a lattice point in another node and import from that node the directional densities of the lattice point associated with the adjacency. Therefore, the total number of sends and total number of receives per update cycle are each equal to at most the number of directions in which the densities flow. When a subcube lies on a face or at the corner of the global lattice, it will not send and receive in all directions. An incorrectly designed schedule of send and receive operations can cause deadlock, and a sub-optimal schedule, such as doing all sends before doing any receives, can cause poor performance.

A Petri net model is a useful way to investigate the liveness and performance of a proposed schedule.

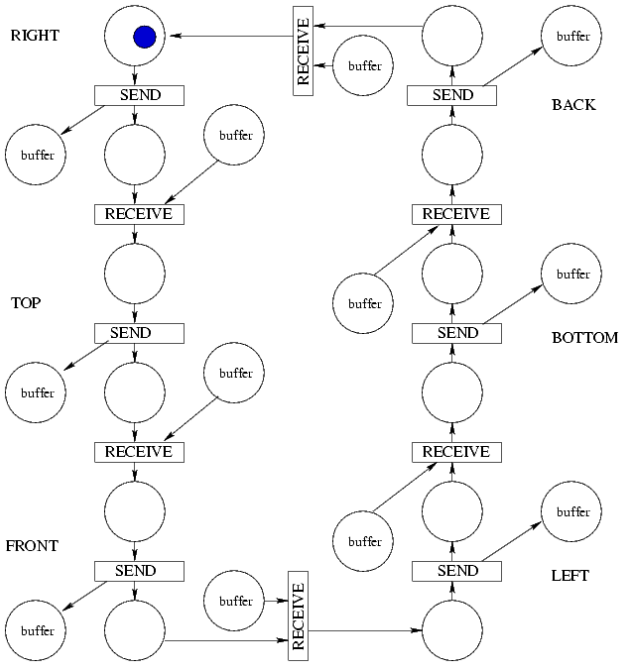

Figure 7: Segment (cube) from the LB Flow Net

In the photon transport model cited above, 18 directions of flow were used, but for simplicity here we consider only face crossing flows in the six directions: $(1, 0, 0)$; $(0, 1, 0)$; $(0, 0, 1)$; $(-1, 0, 0)$; $(0, -1, 0)$; $(0, 0, -1)$. We identify these directions as $d_1, ..., d_6$, and denote a send to an adjacent subcube in direction d_j by s_j and a receive from the subcube located in direction d_i by r_i. A possible communication schedule for an interior subcube consists of any permutation of s_j, r_i for which j and i each take on the values $\{1, 2, 3, 4, 5, 6\}$ exactly one time. The schedule shown in Figure 7 is $\{s_1, r_4, s_2, r_5, s_3, r_6, s_4, r_1, s_5, r_2, s_6, r_3\}$.

This figure shows the operation of a single interior subcube. In the complete Petri net, each subcube is represented by an analogous circular segment. Coupling between the segments is via the places labeled *buffer*. A receive transition cannot fire until the buffer place receives a token as a result of the firing of the corresponding send in the neighboring subcube. Corner, edge, and face subcubes implement a subset of this general schedule. For example, for the subcube located at the origin of the global lattice, the schedule reduces to: $\{s_1, s_2, s_3, r_1, r_2, r_3\}$. For the subcube at the diagonally opposite corner, the schedule is $\{r_4, r_5, r_6, s_4, s_5, s_6\}$.

It can be shown by induction that if all nodes use this schedule, deadlock free communication will be achieved, but the question of schedule optimality remains open. Using this schedule on a Beowulf cluster, we have observed that as the number of subcubes increases from 27 to 125, while the size of a subcube remains constant, the global rate at which exchange/update cycles are completed decreases by over 50%. Since the amount of work done per node is not changing, the reasons for this slowdown are not clear, but they are thought to be related to jitter in the exchange process. A timed version of this Petri net will be of use in gaining a better understanding of this phenomenon.

These nets are completely conflict-free, and so we might expect, a priori, that the parallel simulator would significantly outperform the serial simulators on all Petri nets of this type. The results of executing each simulator on multiple nets are shown in Figure 8. To be consistent with the previous test cases, we have indexed the nets by the number of places that appear in all subcubes. Place count always exceeds transition count for these nets. Thus, for example, a $3 \times 3 \times 3$ arrangement of subcubes has 324 places and 216 transitions. It is then indexed as 324. Again, 95% confidence intervals about each of the mean values shown were computed, but the largest interval for any point was 0.461262, and so they have been omitted from the figure.

The results are almost as expected. The parallel simulator wins easily for large nets, and the crossover occurs at an earlier point (for total places) than in the dining philosophers case. The only surprise is that the growth rate in execution time for the parallel simulator has now dropped to near zero over the measured range.

5 DETAILS AND LIMITATIONS

Although the strong potential for using graphics hardware in Petri net simulators is clear, many issues remain to be resolved. Perhaps most important are the performance implications of allocating various components of the overall task between CPU and GPU. The cost of conditionals on the GPU will be an important factor. As an experiment, we wrote a fragment program wherein each fragment looked up its horizontal and vertical neighbors' colors (toroidal boundary conditions), averaged them, and then wrote the average as its own color value. For this experiment, we used the more conventional, 32-bit integer, RGBA pixel values. We modified the program to conditionally write the average if the red component value in the neighbor to the north exceeded the green component value in the neighbor to the south. Otherwise it wrote half the average. Even though most of the *Cg* instructions involved neighbor lookups and computation of the averages, adding the single conditional increased execution time by 10% on the Nvidia 6800 Ultra. This is a significant penalty, but it is a vast improvement over the previous architecture. On the slightly older, Nvidia FX 5950 execution time increased by more than 150%.

Another constraint is pixel depth. As the simulator expands to support additional variations on Petri net semantics, it is not clear whether 32 bits per pixel will suffice to carry the required information for each place/transition. Since the update information is cycled to texture memory

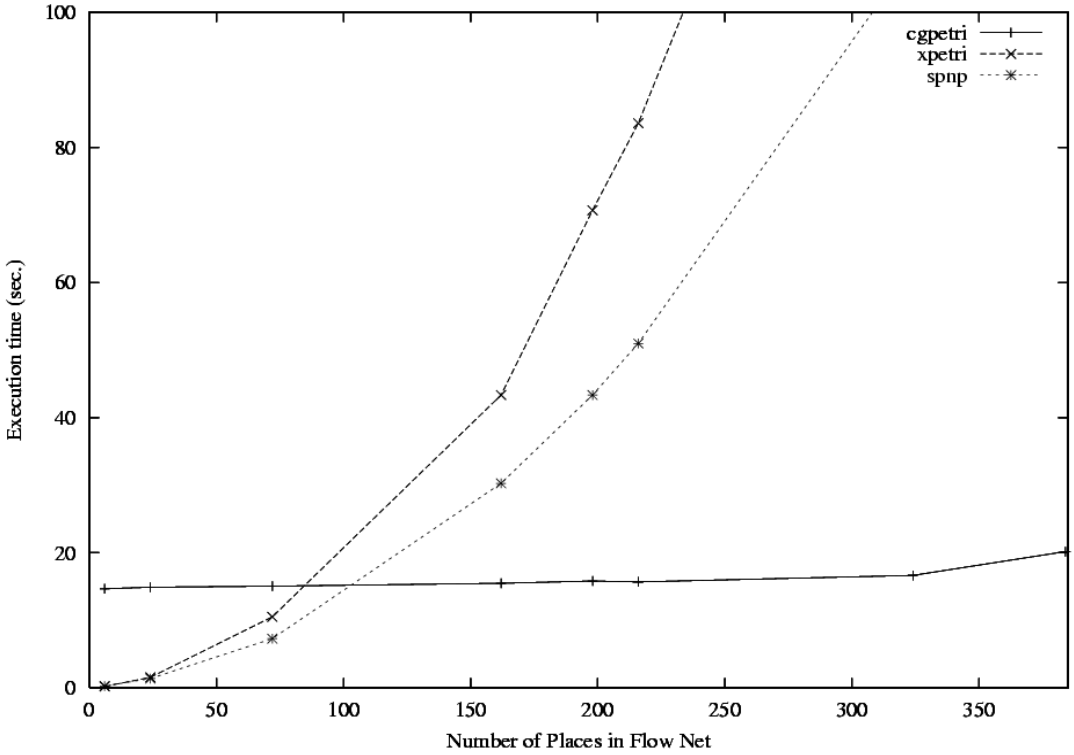

Figure 8: Lattice-Boltzmann Flow Net Execution Time

through the frame buffer, place/transition encodings may have to expand to multiple pixels with the attendant cost. Deeper pixels, up to 128 bits, are available for *pbuffers*, but their use also invokes significant cost.

Basic net semantics are another point of concern. Consider the pair of cycles shown in Figure 9 with net marking (UP1,UP2,DOWN1,DOWN2) = (1,1,0,0). A parallel (in real time) simulation of this non-timed net mandates next marking (0,0,1,1), but classical semantics (Agerwala 1979) permit arbitrarily long loops, i.e., the next marking could be either (0,1,1,0) or (1,0,0,1), and the subsequent marking either (0,0,1,1) or the original. The semantics we have used are right-continuous in the following sense: let $P(N,d)$ denote the process obtained by assigning deterministic firing of length d to every transition in net N. Then $lim_{d\to 0}P(N,d) = P(N,0)$. If alternative semantics are required, the benefits of this SIMD type of parallelism may be lost.

We have not added timed transitions to our simulator design, but doing so appears to be straightforward. Because most of our applications are more easily described by models that permit "token stealing", we restrict our focus here to *atomic firing* and exclude *preselection* semantics (Ferscha 1994). The global clock and event queue will be managed on the C/OpenGL side, and the main loop will advance by event times. Transitions will be encoded as:

value	meaning
0.0	not enabled
1.0	enabling now
2.0	timer in progress
3.0	ready to fire now

The call to *enable()* will receive only transitions of codes 0.0 and 2.0; it will effect changes 0.0→1.0. The call to *count_reservations()* will include transitions of types 1.0 and 2.0. Back on the C/OpenGL side, timer expiration events will be processed with 2.0→3.0, and changes 1.0→2.0 will be accompanied by distribution samples and insertion of expiration events. Conflict resolution will include types 3.0 and 2.0, with the former given precedence over the latter in token stealing. The call to *fire()* will consider only types 3.0, as before.

Adding inhibitor arcs will require almost no changes.

6 CONCLUSIONS

We have proposed a design for parallel simulation of Petri nets on commodity components, the GPUs typically found in desktop PC graphics cards. We have offered a specific design framed in *Cg*, the high-level language provided by Nvidia for programming their 5-series and 6-series cards. Tests of our prototype simulator on Petri nets that are conflict-intensive and on those that are conflict-free consistently

show significant performance gains, over serial simulators, for large nets.

Many issues remain to be resolved. The most pressing is the optimal allocation of software components between CPU and GPU. It is entirely conceivable that alternative designs, with different workload allocation, would show performance gains far beyond those substantial ones that we have reported here. How to systematically seek an optimal design and how to recognize one remain important open questions. It is ironic that Petri nets themselves may be an important tool in addressing these questions.

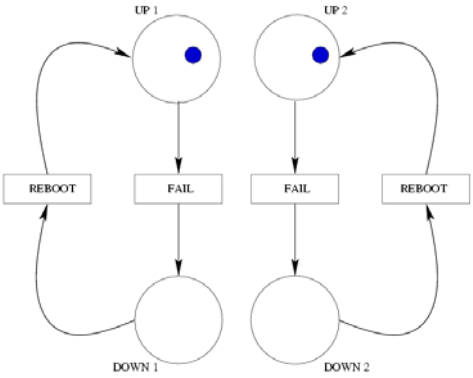

Figure 9: Parallel Cycles

Our overriding goal with this effort is not the design and implementation of a new Petri net simulator. Rather, it is to suggest to authors of commercial-grade simulators that inclusion of a facility for parallel execution on desktop graphics cards would be a worthwhile addition to their products. We note that although CPU development continues to follow Moore's Law (doubling in speed every 18 months), GPU development has shown a sustained rate of 3 times that, doubling every 6 months. Should these trends continue, efforts to take full advantage of GPU capability will become increasingly important.

ACKNOWLEDGMENTS

This work was supported in part by the U.S. National Science Foundation under awards EIA-0305318 and ACI-0113139.

REFERENCES

Agerwala, T. 1979, December. Putting petri nets to work. *IEEE Computer*:85–94.

Baccelli, F., and M. Canales. 1993. Parallel simulation of stochastic petri nets using recurrence equations. *ACM Trans. on Modeling and Computer Simulation* 3 (1): 20–41.

Bolz, J., I. Farmer, E. Grinspun, and P. Schrőder. 2003. Sparse matrix solvers on the gpu: conjugate gradients and multigrid. *ACM Trans. Graph.* 22 (3): 917–924.

Buck, I., and P. Hanrahan. 2003, July. Data parallel computing on graphics hardware. In *Graphics Hardware 2003 Panel: GPUs as Stream Processors*. San Diego, CA. http:// graphics.stanford.edu/ ~ianbuck/GH03_datapargfx.pdf.

Chandy, K. M., and J. Misra. 1981. Asynchronous distributed simulation via a sequence of parallel computations. *Comm. of the ACM* 24 (4): 198–206.

Choi, H., V. Kulkarni, and K. Trivedi. 1993b, June. Transient analysis of deterministic and stochastic petri nets. In *Proc.* 14^{th} *Int. Conf. on the Application and Theory of Petri Nets*, 166–185. Chicago, IL.

Choi, H., V. Kulkarni, and K. Trivedi. Rome, Italy, September, 1993a. Markov regenerative stochastic petri nets. *Proc.* 16^{th} *IFIP Int. Symp. on Computer Performance Modeling, Measurement, and Evaluation*:339–356.

Fernando, R., and M. Kilgard. 2003. *The cg tutorial*. Boston, MA: Addison Wesley.

Ferscha, A. 1994, December. Concurrent execution of timed petri nets. In *Proc. of the 1994 Winter Simulation Conference*, ed. J. Tew, S. Manivannan, D. Sadowski, and A. Seila, 229–236. Orlando, FL: Soc. Comp. Sim. Int.

Geist, R., D. Crane, S. Daniel, and D. Suggs. 1994, December. Systems modeling with xpetri. In *Proc. of the 1994 Winter Simulation Conference*, ed. J. Tew, S. Manivannan, D. Sadowski, and A. Seila, 611–618. Orlando, FL: Soc. Comp. Sim. Int.

Geist, R., K. Rasche, J. Westall, and R. Schalkoff. 2004, June. Lattice-boltzmann lighting. In *Rendering Techniques 2004 (Proc. Eurographics Symposium on Rendering)*, 355 – 362,423. Norrkőping, Sweden.

German, R., and C. Lindemann. Rome, Italy, September, 1993. Analysis of stochastic petri nets by the method of supplementary variables. *Proc.* 16^{th} *IFIP Int. Symp. on Computer Performance Modeling, Measurement, and Evaluation*:320–338.

Goodnight, N., C. Woolley, G. Lewin, D. Luebke, and G. Humphreys. 2003, July. A multigrid solver for boundary value problems using programmable graphics hardware. In *Proc. Graphics Hardware 2003*, 102–111. San Diego, CA.

Hanrahan, P. 2004, August. Stream programming environments. In *ACM Workshop on General Purpose Computing on Graphics Processors*, A–4. Los Angeles, CA.

Hirel, C., B. Tuffin, and K. Trivedi. 2000, March. SPNP: Stochastic petri nets. version 6.0. In *Computer Performance Evaluation, Modelling Techniques and Tools (TOOLS 2000), Lecture Notes in Computer Science, Volume 1786*, 354–357. Schaumburg, IL, USA: Springer Verlag.

Marsan, M., and G. Chiola. 1987. On petri nets with deterministic and exponentially distributed firing times. In *Lecture Notes in Computer Science*, Volume 266, 132–145. Springer-Verlag.

Marsan, M., G. Conte, and G. Balbo. 1984. A class of generalized stochastic petri nets for the performance evaluation of multiprocessor systems. *ACM Trans. on Comp. Sys.* **2**:93–122.

Moreland, K., and E. Angel. 2003, July. The fft on a gpu. In *Proc. Graphics Hardware 2003*, 112–119. San Diego, CA.

Nicol, D., and S. Roy. 1991, December. Parallel simulation of timed petri nets. In *Proc. of the 1991 Winter Simulation Conference*, ed. B. Nelson, D. Kelton, and G. Clark, 574–583. Phoenix, AZ: IEEE Comp. Soc.

Thomas, G., and J. Zahorjan. 1991, December. Parallel simulation of performance petri nets: Extending the domain of parallel simulation. In *Proc. of the 1991 Winter Simulation Conf.*, ed. B. Nelson, D. Kelton, and G. Clark, 564–573. Phoenix, AZ: IEEE Comp. Soc.

AUTHOR BIOGRAPHIES

ROBERT GEIST is a Professor of Computer Science at Clemson University. His current research interests include distributed rendering, rendering participating media, and GPU-based simulation and optimization. He received a Ph.D. in mathematics from the University of Notre Dame. His e-mail address is <rmg@cs.clemson.edu>.

JACOB HICKS is a senior undergraduate at Clemson University. Starting in the Fall 2005, he will attend graduate school in computer science at the University of North Carolina at Chapel Hill. His research interests include GPU programming, rendering algorithms, and physically based rendering. His e-mail address is <jacobh@cs.clemson.edu>.

MARK SMOTHERMAN received the Ph.D. degree in computer science from the University of North Carolina at Chapel Hill. He is presently Associate Professor of Computer Science at Clemson University. His current research interests include computer architecture and graphics hardware. His e-mail address is <mark@cs.clemson.edu>.

JAMES WESTALL received the Ph.D. degree in mathematics from the University of North Carolina at Chapel Hill. He is presently Professor of Computer Science at Clemson University. His current research interests include distributed rendering, computer network performance analysis, and automated tools for CPU/GPU task allocation. His e-mail address is <westall@cs.clemson.edu>.