View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by NASA Technical Reports Server

1995 115430

COLOR ILLUSTRATIONS

Scheduling Time-Critical Graphics on Multiple Processors *

N95- 25870

Tom Meyer and John F. Hughes
NSF/ARPA Science and Technology Center for
Computer Graphics and Scientific Visualization,
Brown Site
{twm,jfh}@cs.brown.edu

44086
P. 5

Abstract

This paper describes an algorithm for the scheduling of time-critical rendering and computation tasks on single- and multiple-processor architectures, with minimal pipelining. It was developed to manage scientific visualization scenes consisting of hundreds of objects, each of which can be computed and displayed at thousands of possible resolution levels. The algorithm generates the time-critical schedule using progressive-refinement techniques; it always returns a feasible schedule and, when allowed to run to completion, produces a near-optimal schedule which takes advantage of almost the entire multiple-processor system.

CR Categories:

Additional Keywords:

1 Introduction

Scientists who create complex datasets (e.g., large time-varying fluid-dynamics simulations, high-resolution MRI scans, and structural simulations) require correspondingly sophisticated ways of examining and visualizing this data. In such cases, a scientist may want to interactively manipulate and examine very complex visualizations, such as a rake with dozens or hundreds of streamlines. Maintaining the fast interaction rates required can be quite difficult, especially in immersive environments such as the Virtual Wind Tunnel at NASA Ames [BL91], where update rates of at least ten frames/second are required.

In order to support the task of exploratory visualization in these complex datasets, we have developed a time-aware scheduling algorithm to provide importance-based real-time computation and rendering of some common scientific-visualization techniques. This algorithm takes advantage of a dedicated graphics workstation with a single-threaded graphics pipeline and from one to several dozen processors, communicating using a shared-memory model. (Although some research systems, such as UNC's PixelFlow, will use multi-threaded graphics pipelines, no such system is commercially available yet.)

The techniques described in this paper extend to general graphics scheduling. Objects with nearly-continuous representations—streamlines can be computed at arbitrary timesteps, and for any number of steps, for example—are particularly well-suited to be scheduled using this algorithm.

This scheduling algorithm has the following advantages

- After an initial startup phase, it can terminate at any time during its incremental refinement phase, and will always return a feasible schedule.
- It results in near-maximal usage of single- and multiple-processor machines if allowed to run to the completion of the refinement phase.
- It pipelines all computations to be rendered as soon as possible, reducing lag times.
- It balances the benefit of spending time computing new data against the time required to redisplay existing data at its already-computed resolution.

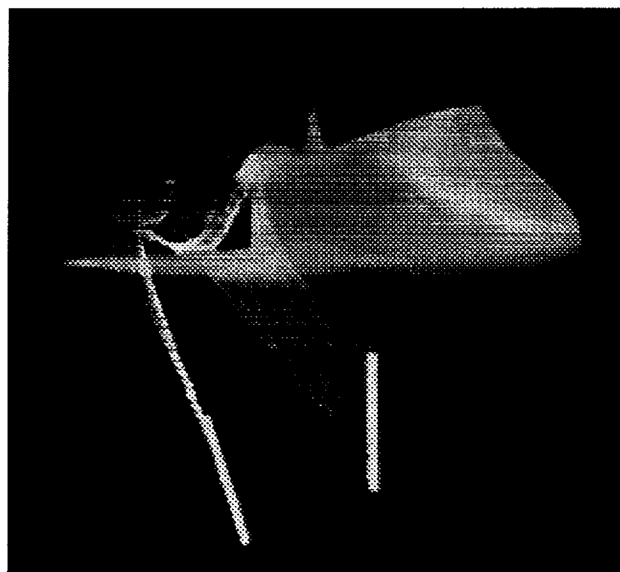

Figure 1: The target application, scientific visualization of complex computational fluid dynamics scenes in an immersive environment.

2 Previous Work

An large body of research on real-time scheduling exists, dating from the 1950's. A good introduction to the relevant issues is [SSNB94].

Classical real-time theories mainly deal with static scheduling problems in which the algorithm has complete knowledge of the demands placed on it, and where there are generally hard constraints

*This work was supported in part by grants from NASA grant NAG 2-830, NSF/ARPA, Sun Microsystems, Autodesk, Microsoft, and TACO.

which, if violated, could result in catastrophic failure (airplanes crashing, factories blowing up, etc). These types of problems are fairly well-understood, and many algorithms exist for solving them.

Although dynamic multiple-processor scheduling is becoming an active area, little work has been done on it. Almost all the interesting problems are NP-complete, and good approximation algorithms are only beginning to emerge.

Unfortunately, problems of particular interest to scientific visualization have not been much studied. The narrowness of the problem – scheduling multiple independent tasks-pairs (computation and rendering) with the two requirements that the "rendering" portions all take place on one machine, and that each rendering portion start only after the completion of its computation portion – makes it too specialized to warrant much attention, except from those who need solutions.

Because virtual environments require near-constant, high frame rates, several systems which address time-critical issues have been developed:

Richard Holloway's Viper system [Hol92] uses objects with predefined levels of resolution, and renders objects at a global level of resolution sufficient to display all of them in allotted time. It does not provide for individual levels of importance for the objects.

Funkhouser and Sequin describe a real-time scheduling algorithm for complex virtual walkthroughs in [FS93]. However, their algorithm only provides support for objects with a few pre-computed levels of representation and their faster algorithm only works well on objects with a convex-downward benefit function. Nonetheless, the ideas in that paper provide the starting point for our algorithm.

The multi-processing scheduling algorithm described by Rohlf and Helman in [RH94] schedules computation, culling and rendering of geometric data by using pipelining, which results in the addition of one frame's worth of lag to the system (lag is as bad as low frame rates in virtual environments). Additionally, they use a feedback-based model for managing scene complexity, so cannot bound frame times when the scene changes rapidly.

Little work has been done on combining computation and rendering, and on managing tradeoffs among computing expensive but useful information.

3 Benefit Function

For a set of rendering tasks, we need to determine the most useful amount of time to spend computing and rendering each one. We define a function *Benefit*(t), which reflects the approximate value of spending an amount of time t computing and rendering a graphics task. We want to maximize

$$B = \sum_i Benefit_i(time_i)$$

subject to

$$\sum_i time_i \leq frametime$$

This function consists of a product of several other benefit values, computed on a per-item basis. For any item i, we decompose the benefit of alloting time t to rendering that item into a product of three parts:

$$Benefit_i(t) = Importance_i \cdot Processor_i(t) \cdot Hysteresis_i(t)$$

Importance is an importance value for the item, expressing the object's inherent value, closeness to the viewer, user interaction, and the visual focus of the viewer. Any number of perceptually-based metrics could be weighted into this; in the current implementation we assign high importance to streamlines with which the user is interacting. Defining useful metrics for determining an object's importance, both in perceptual and semantic terms, is beyond the scope of this paper.

Processor expresses the amount of value to be gained by spending that amount of startup, computation, and rendering time. We assume that this is a nondecreasing function, convex to the right of the "startup time," reflecting the idea that for most visualizations, "something is better than nothing, but fine detail is worth only a little more than coarse detail." Our implementation uses $\sqrt{t - t_s}$ for $t > t_s$, where t_s is the startup time, and 0 for $t \leq t_s$.

Hysteresis term is a sigmoid function designed to encourage inter-frame continuity. It varies smoothly from a value of 1 at some point $t < t_{prev}$ up to a value of $1 + \delta$ at t_{prev}, where t_{prev} is the time allocated to the task in the previous frame.

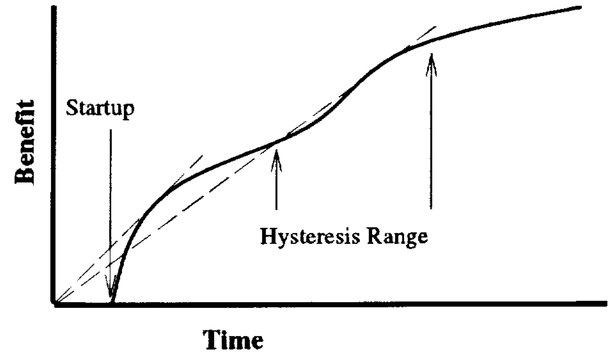

Figure 2: The benefit function has a fixed startup cost, rises quickly after that, and gradually falls off until the hysteresis point. The two dashed lines indicated the local maxima of the function $Benefit(t)/t$.

Since we want to maximize the sum of the benefits subject to the constraint that the sum of the times scheduled for all tasks is less than the frame time, we first examine the benefit per time unit for the tasks. If this benefit per time $\frac{B_i(t)}{t}$, which Funkhauser and Sequin called the *value*, is increasing at some value t, we would ideally like to allocate more time to task i: doing so would reduce the average cost of the benefits derived from executing task i. But if the benefit per time is decreasing, then allocating more time to task i will increase the average cost of the benefit, and should be done only if other tasks cannot benefit more from being given the additional time instead. We therefore consider the points where the value $\frac{B_i(t)}{t}$ is at a maximum as good starting points in the search for ideal time allocations. A complete rationale for this starting choice is given in [FS93]. As seen in Figure 2, the benefit/time function will have at most two local maxima – near the extreme points of *Processor*$_i(t)/t$ and (*Hysteresis*$_i(t) - 1)/t$. If these functions are expressed analytically, it is simple to compute these two points analytically. Note that as long as both functions are differentiable and have few local maxima, we can perform a similar analysis to obtain the starting points.

The functions that constitute *Benefit*(t) are all defined only at a fixed, sparse set of values for t, since computation time cannot be allocated in quanta smaller than the clock cycle. Furthermore, the functions are likely to be step functions, constant on large intervals in the domain. To the extent that this is true, time spent in making small adjustments to the allocation of processor time is often wasted. On the other hand, by bounding the smallest step size we will take in adjusting processor allocations be of the same scale as the smallest interval on which the true benefit functions are constant, we can substantially avoid such waste, while still derviving the benefit of being able to use differentiable functions.

4 Scheduling Algorithm

The application into which the scheduler fits works as follows: In a typical frame-time, user input is gathered, the schedule computed during the last frame is executed, and the schedule of tasks to be performed during the next frame is computed, with the components of each benefit function modified according to the previous schedule (which influences the hysteresis factor) and the user interaction, which influences importance. This sections describes how the scheduler works.

4.1 Single-Processor Case

We use a two-phase incremental-refinement scheduling algorithm, based partly on Funkhouser and Sequin's algorithm [FS93].

Greedy Phase. The first phase is essential; it generates a feasible but not necessarily good schedule, and requires $O(n \log n)$ time. This makes it possible to bound the worst-case time of the scheduler and consider that amount of time as part of the frame time. (The scheduler can place itself into the generated schedule for the next frame). Of course, it is possible to have an extremely complex scene for which it would be impossible to execute even this phase during the frame time. In this case the schedule could be recomputed only once every few frames, at some loss of responsiveness.

The greedy initial phase generates, for each task i that has been selected for inclusion in the schedule, a pair $(t_{ij}, Benefit(t_{ij}))$ at each of the local maxima of the $Benefit(t)/t$ function. We sort these pairs by value $Benefit(t)/t$, and repeatedly take from the list the task whose value is greatest. If the task has not yet been scheduled and there is still available space, we add it to the work list; if it has been scheduled and the new value of t is greater than the previously scheduled one, we reschedule it at the new time (if there is space). This produces an initial packing which is at least half as good as the result from doing the NP-complete optimal packing[FS93].

Incremental Phase. During the second phase of the algorithm the scheduler iteratively refines its generated schedule, as time allows. However it can terminate at any time, since the feasibility of the schedule is never violated. This phase also has $O(n \log n)$ complexity, and may terminate before the time alloted.

The problem reduces to an N-dimensional gradient descent, where we try to find the maximum value of the derivative such that all of the benefit functions have the same slope (some may have a zero slope).

To solve this, we use a version of the Newton-Raphson technique to initialize each of the tasks with a stepvalue δ_i:

$$\delta_i = granularity \cdot \min(1, \kappa - \frac{Benefit_i'(t_i)}{Benefit_i''(t_i)})$$

where $granularity$ is the starting value for the refinement, and κ is the current mean of the derivative values. If δ_i is smaller than the current smallest task size, we initialize it to the smallest task size. This technique moves slowly through areas where the benefit function has high curvature and quickly otherwise, by taking a step that's inversely proportional to the curvature, but proportional to the difference between the current derivative value and the desired derivative value.

While we can feasibly add a task i, we do so, generating a benefit of $Benefit_i'(t_i)$. We add δ_i to t_i, and then repeat. If we cannot feasibly add any task, we find the currently-scheduled task i with the smallest $Benefit_i'(t_i)$, we subtract δ_i from the time allotted to it. We then repeat the entire process. Every time a task has time added to it and then subtracted away, we halve the value of δ_i, until δ_i is smaller than the smallest task size, at which point the task i is removed from the list of candidates for improving the schedule. The algorithm terminates when the list is empty, or the available time is used up.

To determine if we should spend additional time refining the schedule before executing it, we compare the margin of change in the total benefit per iteration with the amount of time required to perform that iteration. If the scheduler should run for less time, we decrease its alloted time slightly, bounded by the worst-case time. Otherwise, we can increase its alloted time.

4.2 Multiple-Processor Case

Dedicated graphics multiple-processor workstations are becoming common, especially for high-end scientific-visualization applications. These machines allow light-weight processes which communicate using low-overhead shared memory and synchronization primitives; however, the rendering pipeline is fed from only one processor at a time.

Most multiple-processor scheduling algorithms are NP-complete (even the fairly simple case of 2 processors, no precedence constraints, and arbitrary computation times is NP-complete) [GJ75]. Since we want to schedule a set of tasks on several processors with precedence constraints, our problem is at least this hard.

We modify the single-processor greedy algorithm to quickly generate a feasible schedule for multiple processors in $O(n^2)$ time for a guaranteed schedule or $O(n \log n)$ time for an optimistic, probably-feasible one.

First, let us consider how we might build a good multiple-processor schedule. Generally we have two portions of the visualization task: a compute task taking time c and a render task taking time r (possibly with a cull task inserted between them). The compute task can run on any processor, but all render tasks must stay together. Also, any data must be computed before it can be rendered, so any good schedule would have to make sure that rendering tasks would not sit idle while waiting for data. Let us consider two tasks which are being computed on a single processor and rendered on another. If we order them so that the tasks with large values of $c - r$ (which we call the *excess compute time*) are last, we have the most room possible for additions, and minimize the startup differences and ending differences between the processors, as shown in Figure 3.

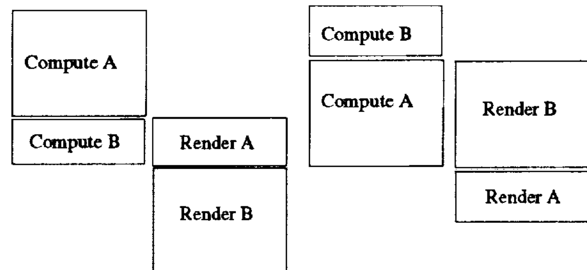

Figure 3: Ordering tasks by increasing excess compute time minimizes the makespan.

The multiple-processor algorithm works like the single-processor algorithm, except that we modify the insertion routine to verify that adding work to the schedule doesn't produce an infeasible schedule. We initially try to add each task to the rendering processor; if there is not enough room on a processor, we push tasks onto the next processor, starting with the task with the most excess compute on the current processor. In this way, we always minimize the total amount of computation time required before rendering can begin. Pushing a single task may cause a cascade of pushes, as shown in Figure 4, but we do not attempt to push a task again if a previous

push on that task has failed (there is probably still not enough room for it, since pushes only go in one direction), so we perform at most $O(p \cdot n)$ pushes (p is the number of processes), with each attempted push requiring an additional verification pass.

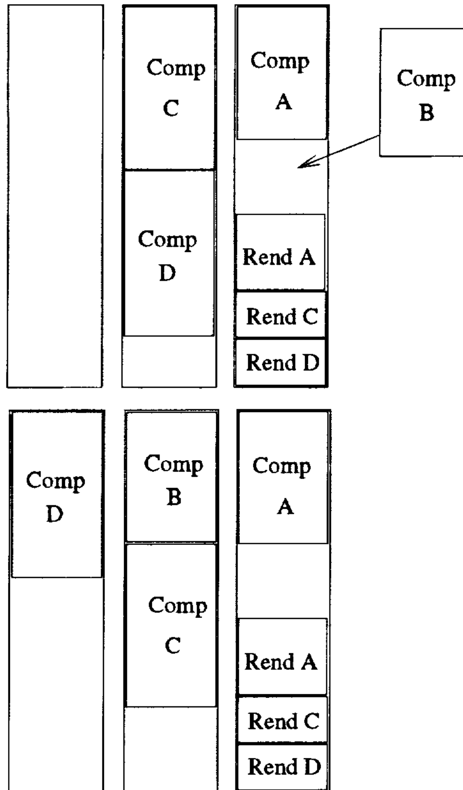

Figure 4: Pushing a task may cause a cascade of pushes. Here, inserting task B causes task D to move to another processor.

In order to guarantee feasibility, we need to look at the two possible ways in which an insertion could violate it:

- We must make sure that the sum of the work is less than the frametime, for each processor. Verifying schedule-size feasibility takes $O(1)$ time if done as each task is added to the list.
- Additionally, indivisible tasks cannot be rendered in time less than the sum of the startup, compute, and rendering times for that task. Consider a fine-grained compute task that generates small pieces of geometry (meshes, lines, or even individual polygons and line segments) at regular intervals c during computation, after the startup time s. Rendering of any of that task's data cannot begin until time $s + c$. If the time required to render a piece is r, the total time required to render x primitives is $s + c + (x - 1) \cdot \max(c, r) + r$. Any generated schedule which violates this requirement is infeasible.

  Verifying precedence relations takes $O(n)$ time, since it is necessary to check every rendering task which is scheduled before an inserted rendering task, as well as every computational task which is scheduled after an inserted computational task.

As described previously, when scheduling several tasks, we can lower the possibility of feasibility conflicts by ordering them from low to high excess computation. Where this value is equal, we define a consistent ordering of tasks so that the partial order of tasks is identical across both the compute and rendering phases. In actual practice this heuristic, when applied to scenes containing diverse types of objects, results in schedules which rarely violate the precedence relations and which achieve high processor usage. If one can tolerate the occasional slow frame, removing the precedence checking results in an $O(n \log n)$ algorithm.

Of course, in pathological cases, any render-dominated schedule may have only the same benefit as the single-processor schedule, but in most complex and diverse scenes, the scheduling algorithm is able to take advantage of the different computational and rendering demands of these tasks to generate a feasible, good schedule which uses almost the entire multi-processor system.

5 Implementation Results

Our actual implementation results are fairly preliminary at this stage, but we have already found some interesting results.

We used a two-processor Onyx to run our real-time usability tests. However, we were unable to isolate the processors from the vagaries of UNIX scheduling (we could only have isolated one processor from the operating system, since system tasks have to run on at least on processor), so obtained frame rates that varied between 8 and 12 frames per second when we attempted to schedule at 10 frames per second.

If we simulate a four-processor Onyx, allocating 100 milliseconds (one complete processor at 10 frames/second) for the scheduling phase, we fill 97% of the other three processors, when the task is compute-bound. With 50 milliseconds allocated for the scheduling task, the algorithm still manages to fill 92% of the available processor time. The benefit of the algorithm declines sharply after this point, however. Giving the scheduler 25 milliseconds resulted in only 65% processor usage, while 15 milliseconds dropped to 32% usage, resulting in less computational time than a single processor.

We intend to perform additional investigations to understand how the scheduling algorithm's behavior degrades under stress, and modify the search techniques which it uses to cope better with situations involving too much work or too little time to schedule.

6 Future Work

We have not yet completed the interleaved version of this algorithm, in which it schedules iself as a time-critical task onto the work queue, scheduling the next frame while the current frame is being completed. Several interesting problems occur here, involving feedback problems and how to deal with unexpected user input (the scheduler can't tell when a user will manipulate a large, complex visualization tool, so will necessarily lag slightly behind in that case).

It is also be extremely important to understand more about human perception in complex environments, to generate more realistic benefit functions. Additionally, we would like to work with scientists to determine the actual semantic benefit of each tool when performing varying types of tasks.

References

[BL91] Steve Bryson and Creon Levitt. The Virtual Windtunnel: An Environment for the Exploration of Three-Dimensional Unsteady Flows. In *Visualization '91*, pages 17–24, 1991.

[FS93] Thomas A. Funkhouser and Carlo H. Séquin. Adaptive Display Algorithm for Interactive Frame Rates During Visualization of Complex Virtual Environments.

In James T. Kajiya, editor, *Computer Graphics (SIGGRAPH '93 Proceedings)*, volume 27, pages 247–254, August 1993.

[GJ75] R. Garey and D. Johnson. Complexity Results for Multiprocessor Scheduling Under Resource Constraints. *SIAM Journal of Computing*, 1975.

[Hol92] Richard L. Holloway. Viper: A Quasi-Real-Time Virtual-Environment Application. Technical Report TR92-004, University of North Carolina, Chapel Hill, 1992.

[RH94] John Rohlf and James Helman. IRIS Performer: A High Performance Multiprocessing Toolkit for Real–Time 3D Graphics. In Andrew Glassner, editor, *Proceedings of SIGGRAPH '94 (Orlando, Florida, July 24–29, 1994)*, Computer Graphics Proceedings, Annual Conference Series, pages 381–395. ACM SIGGRAPH, ACM Press, July 1994. ISBN 0-89791-667-0.

[SSNB94] John A. Stankovic, Marco Spuri, Marco Di Natale, and Giorgio Buttazzo. Implications of Classical Scheduling Results for Real-Time Systems. *to appear in IEEE Computing*, 1994. `ftp.cs.umass.edu`, `/pub/ccs/spring/impl_sch_rts.ps`.