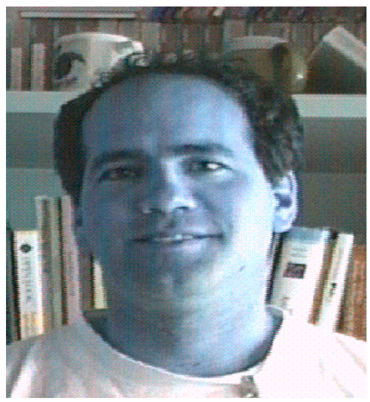
A Case Study in Multithreaded Programming

by Chris Nicholas

To become acquainted with the programming style and system behavior of multithreaded applications, a public-domain synthetic rendering program was modified to run in a parallel, distributed fashion. For computational intensive scenes, nearly linear performance increase was achieved with additional CPUs. A few programming conventions well-suited to multithreaded programming were also adopted and are outlined in this article.

Before reading this article, the author would like to draw attention to Sun document number 801-3176-03, *Guide to Multithread Programming*. This document is located in the source/doc directory in the “MT_pgrm_guide.ps” in postscript format. It contains an excellent discussion of the fundamental issues in multithreaded programming, as well as discussionand examples of the Solaris threads library itself.



Case study: Distributed RVIEW

Synthetically rendering a scene, by individually tracing visible rays from the viewpoint back to light sources (“ray tracing”), is a problem well-suited to parallel solutions. Because of the principle of superposition (i.e., light rays do not interact with each other) each ray can be traced independently and the individual results combined to form an image. The ray tracer selected for this exercise was RVIEW, a viewing program of the RADIANCE public-domain package from the Department of Energy. RADIANCE is used to determine compliance of buildings with federal workplace lighting requirements. Because the program consists of some 90,000 lines of code and represents one of the most complete, mathematically accurate ray tracers anywhere, no attempt is made here to describe the software internals more than necessary. Instead, the focus of this article is to summarize the resource-bound portion of the algorithm, distribution of this portion across as many CPU resources as possible, and concentration of the results back to the user's screen. A complete listing of the RADIANCE source tree and reference documents is included in the /src directory.

Future work on this effort includes:

a. Further isolation of the program into client and server functions. The client, in particular, still links to numerous files that are not necessarily part of the client functionality.

b. Several functions, particularly patterns bidirectional reflection distribution function (BDRFs), are not MT-safe yet.

c. Increased integration with modeling programs and time-sequential movie-loop generation.
d. Multiscreen and steroscopic display extensions.

The design strategy was to modify RVIEW to take full advantage of a local multiprocessor machine. The code was evolved into a generic daemon capable of running on “consenting” machines within the SUN engineering network in a distributed, load-balanced fashion. Because the results are immediately posted to the user's screen, a visual indication of the running computation is produced. This goal was achieved.

A scene is traced in increasingly finer resolution. On the first “pass,” a scene is divided into 256x256 areas and a ray is traced down the center of each area. The entire 256x256 square drawn is the color of the individual ray. The scene is divided into “rectangle lines” of such square areas. Successive passes repeat the process on an increasingly finer scale until the entire scene is traced on an individual pixel basis.

Architecture

Local Multiprocessor Operation

The single machine, multiple-processor architecture is separated into three fundamental thread types:

- The main loop,
- The GUI service thread draw_thread(); and
- The compute-bound ray tracing threads.

Two main global memory buffer areas exist:

- The overall scene description built by main() and read by the newpaint() threads; and
- A group of scanline buffers written onto by ray tracer threads and read by the draw_thread().

One of the primary design goals was to avoid locking on an individual pixel basis. In order to acheive this goal, access to buffer memory is coordinated on a scanline basis; the scanline buffer areas are coordinated by a “free” stack of indexes to unassigned buffers and a queue of indexes of scanlines that are completed and need to be drawn. Additionally, pointers to functions can be placed in the buffer area. Four routines, check_in, check_out, add2q, and drain_q, effectively circulate indexes through the system. This association of index stack, index queue, and unlocked buffer areas is referred to by the author as a memory “recirculation” system.

(See Appendix A)
Figure 1. Local Architecture.

The main() thread:

a. Parses the command line arguments and builds the scene description octree structure.
b. Creates the draw_thread() “service” thread.
c. Creates ray tracer threads for each refinement pass.

The draw_thread():

a. Drains the draw queue of indexes to draw and draws rectangles of appropriate size to the screen.
b. Checks the X input queue for input events (particularly mouse button events) and takes appropriate action.
c. Returns the indexes of buffers it has drawn to the free stack.

A ray tracer thread:

a. Checks out an index into a scanline buffer and a y_start to indicate which rectangle line to ray trace. Because ray tracing threads fetch their own work, they can be said to be self-scheduling.
b. Traces a rectangle line, at the appropriate refinement level, into the buffer indicated by its index.
c. Checks the index into the draw queue for the draw thread to draw later.
d. When there are no more rectangle lines to be done on this pass, a ray tracer thread calls thr_exit(). The last one to do so also sends a cond_signal back to main indicating that all

rectangle lines for this pass have been completed and placed on the draw queue.

Figure 1 indicates the steps taken to trace a scene.

Distributed Operation

The multiple-machine, distributed architecture extends the initial version by substituting a “connection thread” for the individual ray tracer threads. This operation is particularly well-suited for multithreading because the individual connections frequently block during communication with their associated raytracing daemons. As one thread blocks another is free to utilize the communications hardware. On higher-performance configurations which support multiple ethernet and FDDI devices, such as the SPARCserver and SPARCcenter class machines, such communication can actually occur in parallel. Additionally, another thread, called a “subproject manager” is introduced to perform the synchronization of the connection threads between passes. Future versions of the software aim to cluster multiple, simultaneous, active, multiple-workgroups for tracing different perspectives of the same scene, or different scenes altogether, with the ultimate goal being ray-traced movie sequences. The distributed architecture is shown below.

(See Appendix B)
Figure 2. Distributed Architecture.

Multithreaded Programming Considerations

Global Variables

One of the primary principles of concurrent execution is to utilize synchronization primitives as infrequently as possible, and have all active threads fully utilizing their available resources. Because threads operate within a shared address space, global “write” accesses must be performed with caution. Also, since different threads might intentionally access the same routine simultaneously for parallelism, static variables must be used with great care. Therefore, one of the most useful starting places for multithreading existing software is to determine all instances of global and static variable write accesses. The author used the SunPro SourceBrowser™ variable filtering tool extensively in this search.These SunPro tools were also extremely useful for performance profiling in order to determine candidate tasks for concurrent execution.

Window Systems

Most applications need to run some form of window system. By its very nature, the X Window protocol is difficult to breakdown into useful atomic operations, in that typical usage involves state, embodied in the Graphics context. Additionally, most graphical user interfaces (GUIs) are structured around a main loop which processes X events and dispatches these events to appropriate callback routines.

The challenge is to provide a mechanism for this main event loop, easily encapsulated into a dedicated thread, to communicate with other threads of the application, which might generate events as well. In the authors case, typical events consisted of completion of scanline buffer areas. While the poll() and select() system calls waits on the file descriptor to the X event queue, responding to events generated by other threads is not straightforward. At least two approaches can be taken:

The first approach is to effectively loop while waiting for an event in either of the two sources. One can loop inquiring if anything is pending on the window system file descriptor via the XPending() call, and use a cond_timedwait() for a short period of time to determine if anything is pending on an internal queue.

The second approach is to actually communicate with the GUI thread via the filesystem on a FIFO or a named pipe, and have the GUI add that fd into its select/poll usage.

The former approach was

taken by the author, and has quick, interactive response. The client_draw_thread() is created in rviewclient.c and basically spins between attempting to drain the X input queue, in the try_x_event() routine of x11.c, and the draw_q via try_drain_q() of management.c.

State diagrams and thread groups

The most difficult aspect of multithreaded programming is non-determinsm of when threads are executed relative to one another. To ensure correct behavior of the program, a structured approach is required. In general, the author found it essential to adopt a Mealy-Moore state diagram approach to thread interactions for all but the most trivial examples. Most importantly, all thread library interaction utilizing mutex, semaphore, and suspension primitives, beyond initialization necessarily invoked at change of thread state in an explicit state variable. Any interthread signalling changes the state of both the source and target threads.

For example, consider the th_sig.c program provided in directory source/examples. Two threads are created; one outputs a “.”; and the other outputs a “~”. When run on an MP machine, it is possible to visibly see that both threads are being run. However, when run on a UP machine, only the first thread runs initially; this is in keeping with the Solaris threads library policy that threads run until blocking.

This leads to the question: “How can one coordinate groups of threads, and be certain no thread will 'miss' an event, since successful usage counts upon the fact that all intended recipients have reached the appropriate cond_wait()?”

One method is to use a well known construct referred to as a “barrier.” A barrier is initialized to a given integer value and as threads call barrier_wait(), the value is decremented. Upon reaching zero all threads return from the call via cond_broadcast(), and the barrier_count is reset to its initial value. An example of the use of this construct is given in the “my_barrier.c” sample code located in the source/ examples directory.

A second method is to use a construct referred to by this author as a “down counter.” The idea of a down counter is similar to that of a barrier, except that upon reaching zero, control is returned back to a designated “coordinator” thread. The coordinator thread is responsible for resetting the group's counter and issuing the broadcast. This construct allows a thread to inspect results of the last group activity, and perform adaptive load balancing etc. before the group resumes execution. To resume execution, the coordinator thread is responsible for resetting the group's counter and issuing the cond_broadcast(). An example of this technique is given in the “example1.c” program in the source/ examples directory.

These techniques can be used in many instances. For example, a discrete event simulation can create issue “clock ticks” by either explicitly becoming the coordinator of a down_counter or just joining in a barrier. These constructs can easily be extended to form “timed_barriers” and “timed_down_counters” by utilizing the cond_timedwait() synchronization primitives in their internal routines.

The main routines exhibiting coodination of threads groups are located in the rv3client.c file in the modified version of RVIEW. Additionally, example2.c in the example demonstrates heirachial thread coordination using both down_counters and “manager” threads.

The appendices provide the state diagrams for the thread types of the distributed RVIEW implementation.

Signals

Signals and signal handlers are still identified on a process-wide basis. What is under per-thread control, however, is the signal mask and the ability to direct a signal to a specific thread from another thread.

A useful technique for handling

signals on a per-thread basis is by using thread-specific data, accessed through a thread_key_t variable, to identify which signal handler routine to utilize for a specific signal.

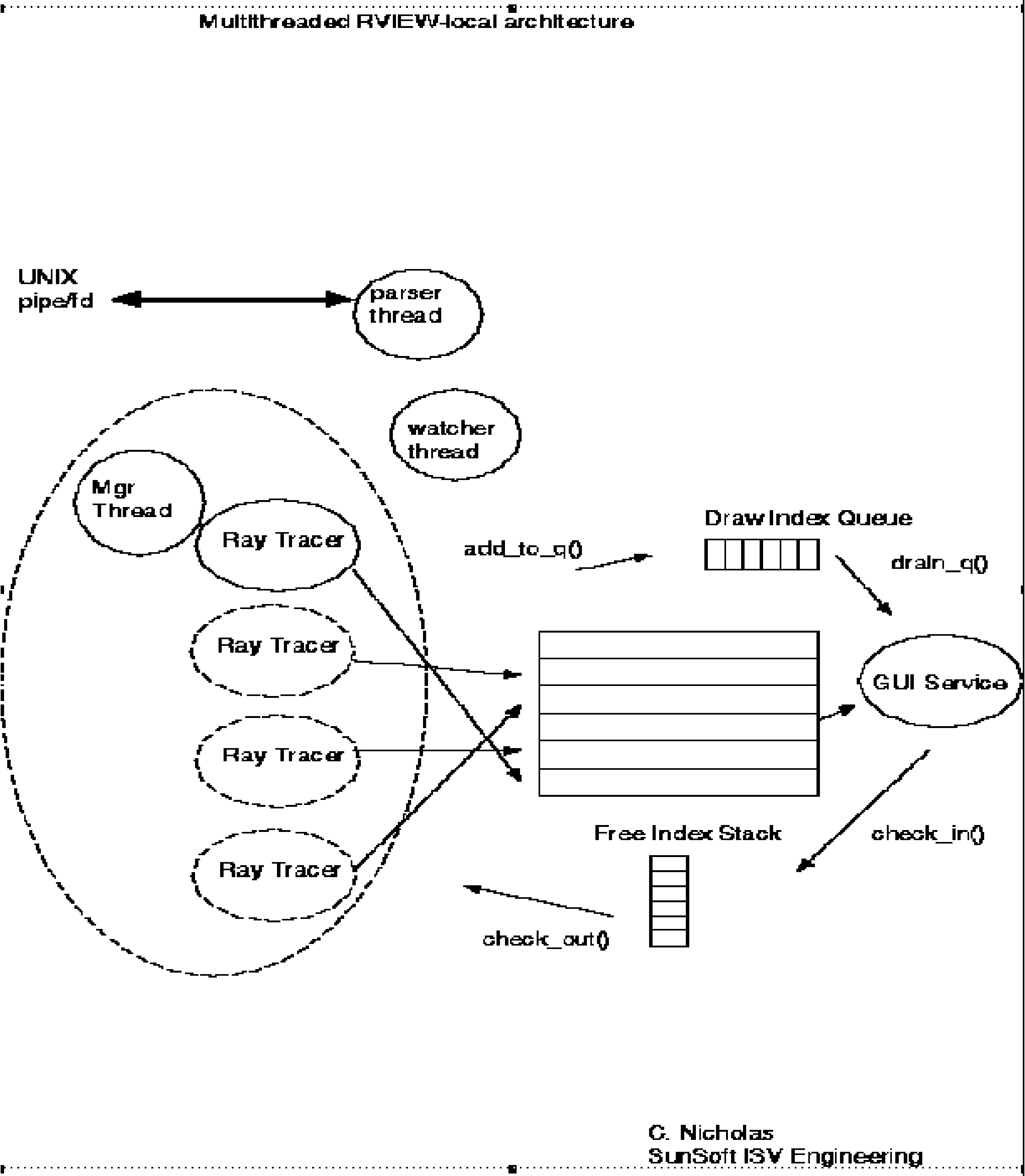
Again, the th_sig.c sample code in the source/examples directory provides an example of such usage.

Conclusion

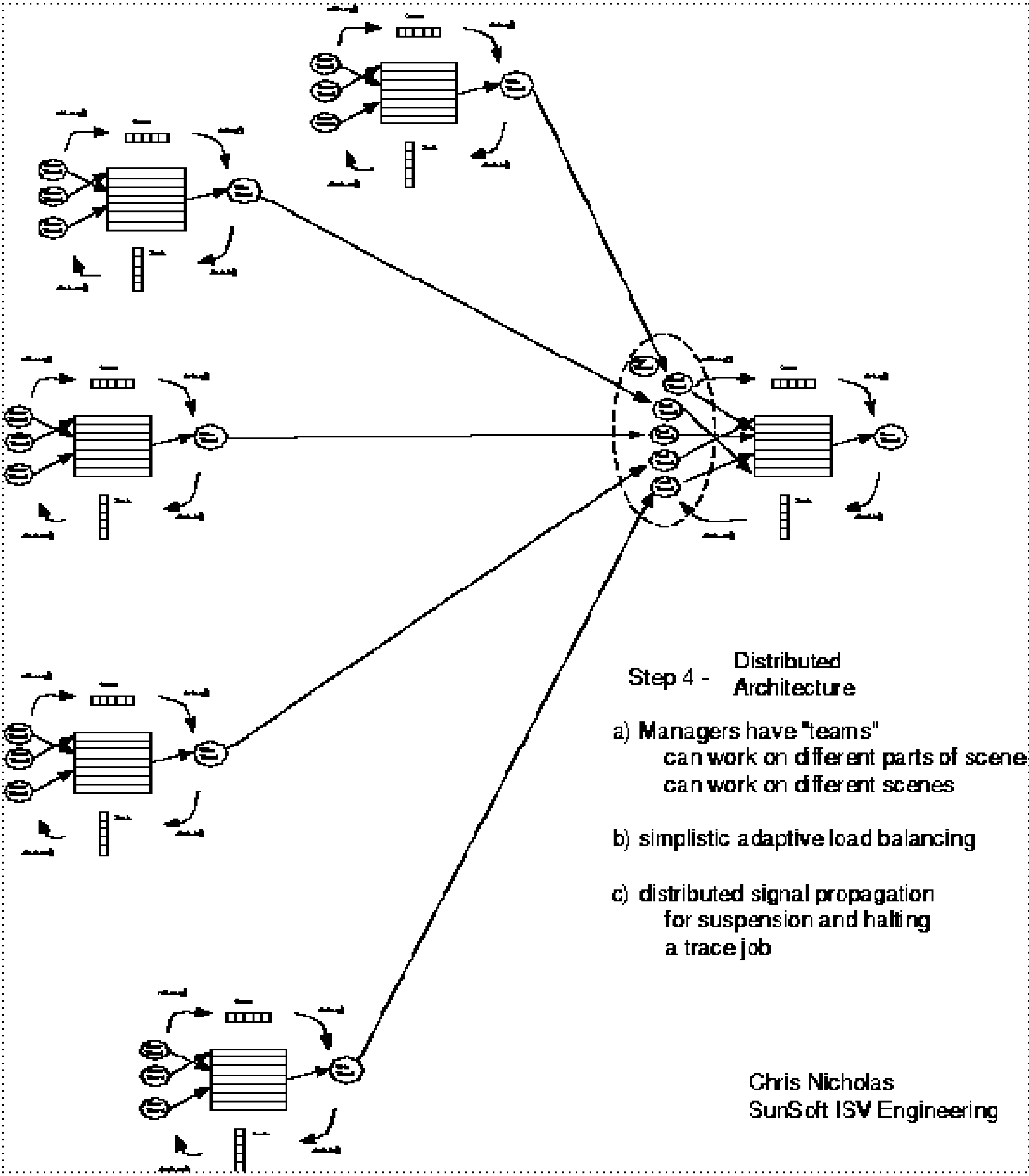
The Solaris user-level threads library provides an elegantmechanism to address today's problems and fully utilize today's multiprocessor architectures. The ability to synchronize cooperating tasks for overall problem solutions promises major advancements in simulation, database, and other emergent technologies of software engineering.

The following books are good resource materials for learning more about multithreaded programming: *Introduction to Parallel Programming* by Steven Brawer (published by Harcourt Brace Jovanovich); *Portable Programs for Parallel Processors* by James Boyle, Ralph Butler, Terrernce Diaz, Barnett Glickfeld, Ewing Lusk, Ross Overbeek, James Patterson, and Rick Stevens (published by Holt, Reinhart and Winston); and *Numerical Algorithms for Modern Parallel Computer Architectures* by Martin Schultz (published by Springer-Verlag).

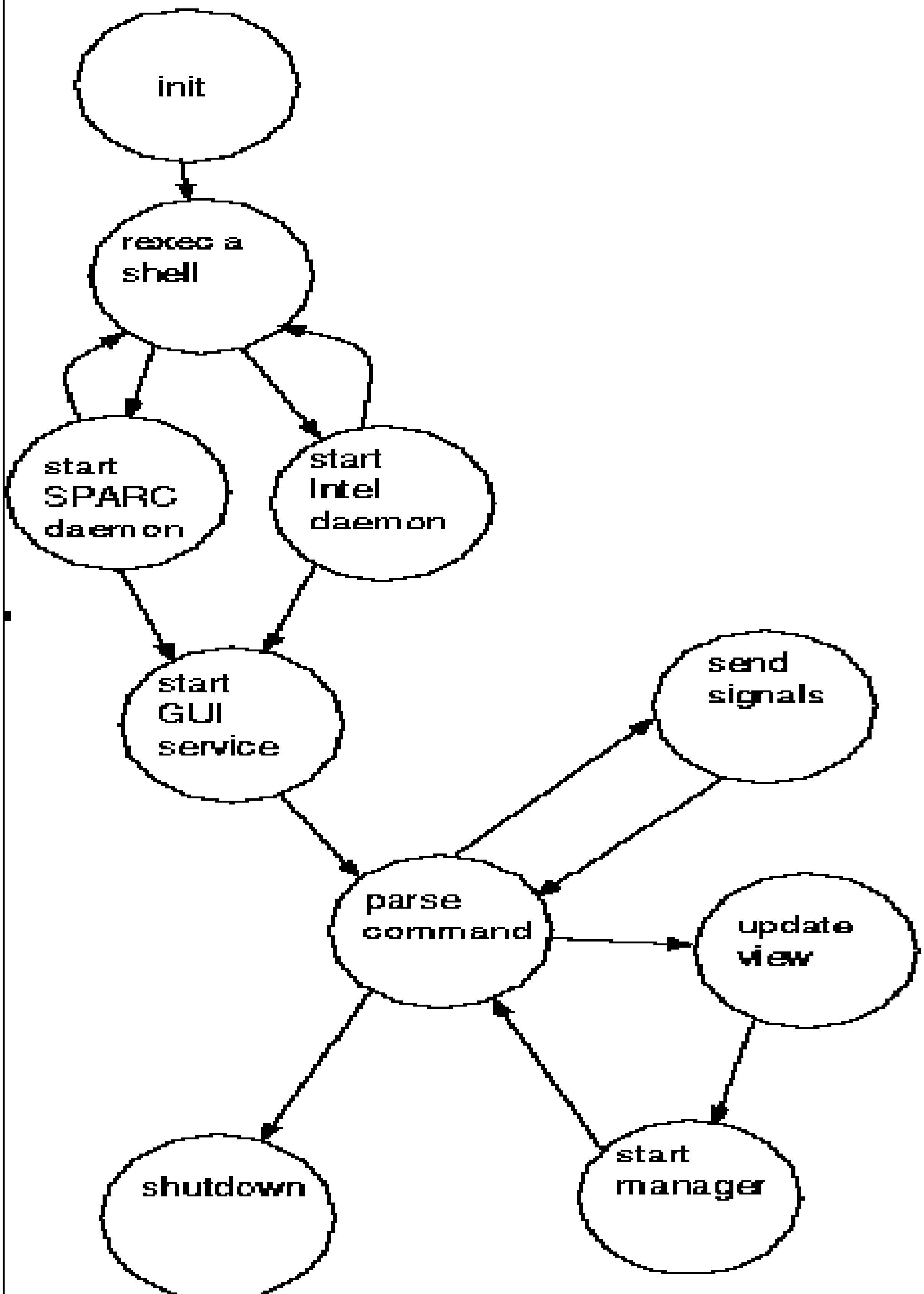
Appendix A (Figure 1 - Local Architecture)

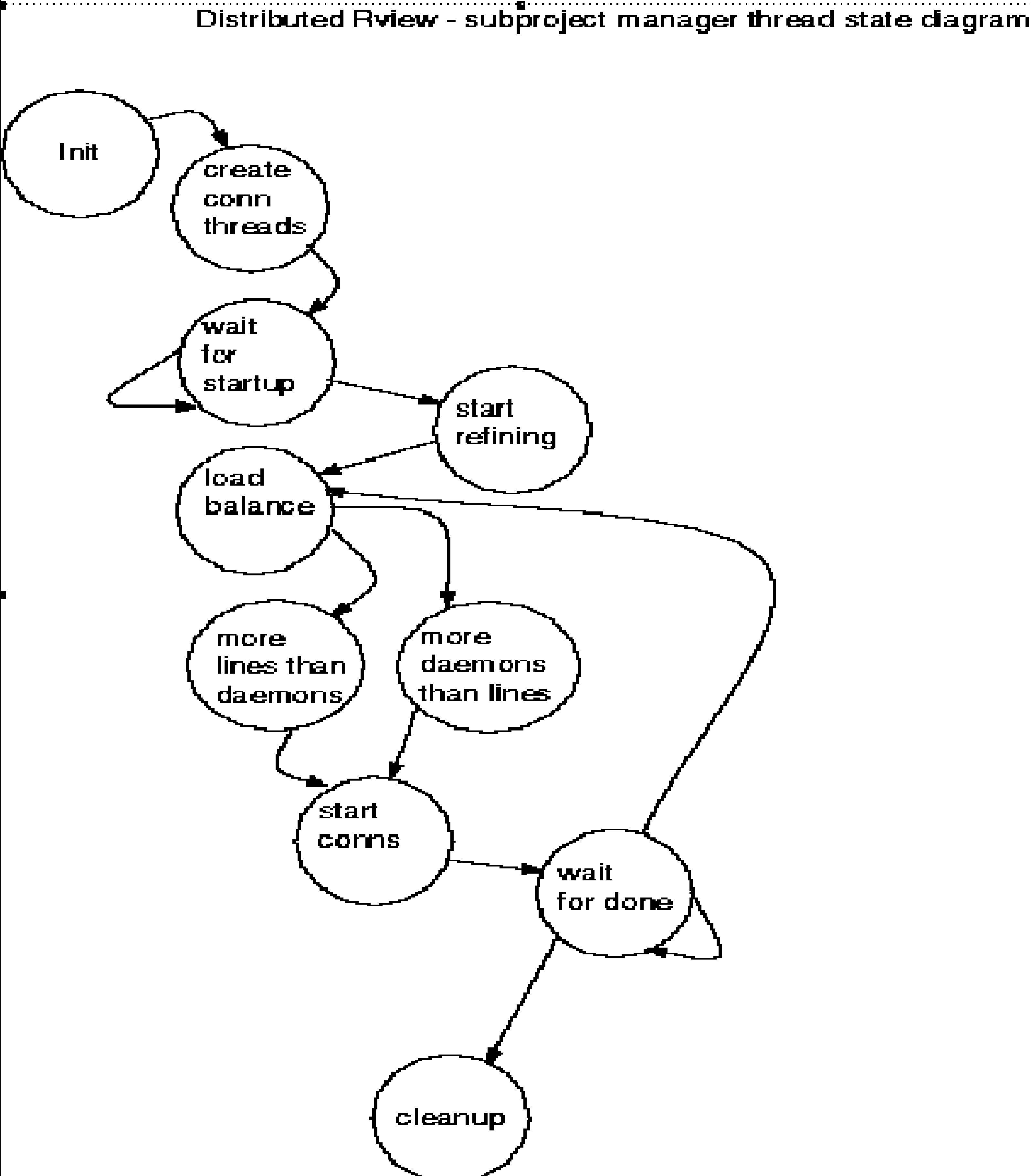


Appendix B (Figure 2 - Distributed Architecture)



Distributed RVIEW - main thread




Distributed Rview - subproject manager thread state diagram
Init
create conn threads
wait for startup
start refining
load balance
more lines than daemons
more daemons than lines
start conns
wait for done
cleanup