



Colorado Technical University

Technical Report

CTU-CS-2001-01

Mark H. Butler, Bo I. Sandén

Scalability of Parallel Discrete Event Simulation Using Threads

Scalability of Parallel Discrete Event Simulation Using Threads

Mark H. Butler

TRW Systems

Bo I. Sanden

Colorado Technical University

The use of simulation as a tool for examining the behavior of both current and planned real-world systems continues to proliferate. The ability to examine the behavior of new complex system concepts, or the effect of modifications to existing ones, before committing large amounts of time and money to implementing them is known to produce high quality, and less expensive, final products. There are many organizations using, or developing, simulation capabilities and TRW is a major participant in many of these simulation projects.

Most large Parallel Discrete Event Simulation (PDES) implementations are implemented on some form of Unix-based platform. They generally use separate Unix processes for event processing which carry with them more overhead than threads. This paper describes an investigation into the use of threads instead of processes for event processing in this type of simulation. The goal was to characterize simulation performance in this environment. The underlying hardware used was the SGI Origin 2000 and multithreading support was provided by the POSIX "pthread" library

Two test programs were created. The first measured the maximum performance improvement (speedup) achievable with 100% thread overlap, regardless of the application. The second test program was a PDES prototype with parameter-controlled runtime characteristics. These included number of threads, number of CPUs, number of events to process, event processing time, and Future Event List (FEL) overhead. Both of these test programs utilized 64 threads allocated over the range of 1-32 CPUs. 64,000 events were run through the prototype with processing time varied from 100 to 1000 microseconds per event. 65 runs of the thread test program, and 1,950 runs of the PDES prototype were made with 1,170 and 42,900 data points collected respectively. Because of intermittent, unpredictable interference on the system being used, runs with each unique parameter configuration were repeated 5 times in order to determine the "best case" performance as closely as possible. The most important data point collected was "elapsed time."

The thread test program results showed that 100% overlap among threads did not necessarily yield perfect speedup. While performance improved very well through the entire range of CPUs, there was some additional overhead incurred as each CPU was added so that, at the 32 CPU level, speedup was about 96% of potential. This could have a significant impact on massively parallel systems with 100s or 1,000s of CPUs. The PDES prototype's performance improved as CPUs were added - up to a maximum of 12-14 CPUs. Beyond that, as CPUs were added, performance remained constant or degraded. The maximum speedup attained within the run profiles tested was ~5. Configurations with the longest event processing times and least FEL overhead showed the most improvement. Conversely, configurations with the shortest event processing times and most FEL overhead showed the least performance improvement.

Areas for further investigation became apparent as a result of this work. One area would provide more precise control over the thread scheduling mechanism by running in privileged, or superuser, mode. Another, while sacrificing some portability, would use the native, platform-specific multithreading support rather than the more generic POSIX pthreads library. A third, and especially interesting possibility, would use multiple FELs - with perhaps multiple threads per FEL.

Introduction

To be filled in with more text, etc. Thought this table would be good for showing the overhead of processes vs threads.

Table 1. Typical Process and Thread Functions - Timings and Ratios

<i>Function</i>	Sun Sparc 4 110MHz		Sun Ultra 1 167MHz		SGI Origin 2000 250MHz	
	µs	Ratio	µs	Ratio	µs	Ratio
Unbound Thread Create	330	1.0	80	1.0	10	1.0
Bound Thread Create	720	2.2	170	2.1	NA	NA
Process Create	45000	136.4	9500	118.8	3139	313.9
Mutex Lock/Unlock	1.80	1.0	0.70	1.0	0.50	1.0
Semaphore Post/Wait	4.30	2.4	3.70	5.3	0.94	51.9
Global Variable Ref.	0.020	1.0	0.006	1.0	0.028	1.0
Thread-Specific Data Ref.	0.590	29.5	0.450	75.0	0.088	3.2

Methodology

Since PDES software has typically been developed in general purpose programming languages, rather than simulation-oriented languages, it was decided to implement this software in a general-purpose language. Modern software engineering practice favors the use of object-oriented technologies and languages, so the decision was further made to use an object-oriented language, if at all possible. The most logical choice was C++. Although Java is an excellent general purpose, object oriented language, its interpreted nature did not lend itself to the high performance requirements of PDES. One possible alternative approach would have been to use ANSI C wrapped in C++.

As with many projects, the build or buy question, with respect to a PDES library, had to be addressed. Developing such a library from scratch seemed to be a formidable task. Additionally, investigation revealed that, while a library that met all the requirements did not seem to exist, several existing simulation libraries could offer a jump start. Two main contenders were identified

The first was the CSIM library, developed and published by Kevin Watkins (Watkins 1993) and written in ANSI C. It had been implemented on a number of different platforms. Some of the compilers on which it had been compiled included the Mix Power C, Borland C, Microsoft C, and GNU gcc compilers. CSIM provided a complete simulation support library providing simulation components such as:

- Executive or Scheduler – responsible for activating the different components of the simulation model in the correct order and for maintaining the simulation clock.
- Entities – the representation in the model of active objects in the simulated domain such as customers, CPUs, etc. An entity responds to events in the model and may cause additional future events to be scheduled.
- Resources – represent some form of reusable assets in the simulated domain such as the amount of free memory in a computer or a checkout stand in a supermarket. The most important characteristic of a resource is its limited capacity.
- Queues – represent waiting lines in front of service facilities. Service facilities are entities that manage resources. Queues occur wherever there is an imbalance between requested resources and the ability of the service facilities to provide those resources.

CSIM had several shortcomings relative to its desired attributes and projected use. First, it was not object-oriented: it had no classes, inheritance, polymorphism, and other characteristics that are the essence of object-orientation. More significantly, it had no built-in multithreading support. A great deal of effort would have been required to restructure the library and add this support. A change of this magnitude would also have exposed the library to new errors introduced by the new structure as well as the new code supporting multithreading.

The second simulation library identified was the HASE++ library, developed by Dr. Fred Howell et al at the University of Edinburgh, Scotland in the 1996-1997 timeframe. It was a multithreaded library created mainly to support the Hierarchical design and Architecture Simulation Environment (HASE), which involved the modeling and simulation of computer architectures. HASE++ was written in C++ and was designed to use an external multithreading library such as POSIX pthreads, Solaris threads, the REX

threading library, or the Cray threading library. HASE++ employed a standard discrete event simulation algorithm consisting of four steps:

- Pop the next event(s) off of the FEL
- Enable the appropriate entity(s) to process the events
- Wait for the entity(s) to finish processing
- Repeat until there are no more events on the FEL

One shortcoming of HASE++ was that its algorithm made it inherently sequential. The only possibility for parallelism was if several events had exactly the same simulation timestamp. All events with the same simulation timestamp would be popped off of the FEL during the first step. These could be run in parallel if several CPUs were available. So, some modification would be required in order to make HASE++ multithread efficiently. The positive points concerning the use of HASE++ were:

- Like CSIM, it was already developed and tested
- It was object-oriented and written in C++
- It had multithreading support built-in

Therefore, the effort required to port the functionality and syntax to the SGI IRIX platform appeared to be a non-trivial, but smaller, task than developing a similar library from scratch or making CSIM object-oriented and/or adding multithreading to it. Its inherent sequential nature was the main obstacle. This was overcome by modifying the FEL management logic slightly to allow all events present on the FEL at any one point in time to be processed. The simulation clock was thus advanced through the times indicated by the timestamps of these events. This modification was made possible by the nature of the experiment driver. More specifically, the experiment driver, `feltest`, created a number of event sources and event sinks (more easily thought of as event generators and event processors). There were always an equal number of event generators and processors and each was assigned to its own thread. A specific event processor was assigned to each event generator. This simplified control of the environment considerably. With this architecture, the simulation ran as follows:

- `feltest` began by creating all of the event generator and processor threads. The number of event generators to be used was specified by the input parameter `num_srcs`. The simulation actually started after all of the event generators and processors were successfully created and waiting on the "start" semaphore.
- When the "start" semaphore cleared, each event generator immediately scheduled an event for its associated event processor and waited on a semaphore for an "event processed" event. At some time in the future when the "event processed" event was received from its event processor, it simulated some processing of its own before scheduling another event for its event processor. These waits were semaphore waits. This process continued until it had generated the number of events specified in the input parameter `num_events`.
- While the event generators started the simulation by generating events, the event processors started the simulation waiting on events. When an event processor was

awakened to process an event, it simulated processing it and then scheduled an “event processed” event for some time in the future. It then waited for the next event. These waits were also semaphore waits.

- The FEL manager thread was the coordinator of all these events. In response to a scheduled event from an event generator, the FEL manager released the event to its respective event processor. When the event processor completed and scheduled an "event processed" event for its event generator, the FEL manager released the event to the associated event generator.

With the above approach, only half of the threads could be running at any one time – the same value as `num_srcs`. This was true because if a specific event generator was running, its associated event processor was waiting for it to schedule an event for it. Conversely, if a specific event processor was running, its associated event generator was waiting for it to schedule an “event processed” event. This approach meant that trying to use a CPU complement greater than half of number of total threads (also equal to `num_srcs`) would provide no additional speedup. Therefore, in the theoretical best-case scenario, there would be two threads assigned to each CPU – only one of which was running at any one time.

The above approach did not change the original design that caused up to `num_srcs` events to be on the FEL at any one point in time. What changed was the logic used for processing them. Rather than just limiting the events processed from the FEL at any point to those with the same, earliest, simulation timestamp, all events on the FEL could be processed and the simulation clock advanced based on them.

It should further be noted that in a more typical simulation, there would not be this 1:1 relationship between the event generators and the event processors. This relationship was used to simplify simulation control. The HASE++ library supports other, less restrictive, approaches. With those approaches, more complex logic would be required to determine how far the simulation clock should be advanced, with the appropriate event processing, at each juncture. Additionally, there would need to be rollback logic for situations where entities posted events with timestamps less than the current simulation clock. This situation was avoided for these experiments by forcing all future events to be scheduled at fixed increments in the future from current simulation time.

The `feltest` simulation prototype program was developed to provide a simulation-like testing tool to examine the effects of multithreading and multiple CPUs on a simulation environment.

The `feltest` program consisted of three separate files written in C++: `main.C`, `src.C`, and `sink.C`. The `src.C` and `sink.C` files also had declaration (`.h`) files associated with them. (Note that, for the SGI MIPSpro C/C++ compiler, the default file name extension for a C++ source code file was an upper case ‘C’ letter. Include file

names had to be fully specified and their extensions, if present, were lower case letter 'h.'). The main.C program was the high-level feltest simulation run control program. It performed four functions:

- Instantiated a HASE++ simulation object with a "sim_system sim;" statement
- Read the run parameters from the input.params file by calling the read_globals() method
- Added the requested number of entities (event generators and event processors) to the simulation by repeatedly calling the sim.add() method
- Caused the simulation to run by calling the sim.run() method

Figure 3 lists the source code for the main.C program:

```
// main.C - FEL test main program

#include "simpp.h"
#include "expt.h"
#include <stdio.h>
#include <unistd.h>
#include "src.h"
#include "sink.h"

int main()
{
    sim_system sim;

    read_globals();

    int num_srcs = get_int_param("num_srcs");

    char src_name[30], sink_name[30];
    printf("main() - adding %d entities (%d srcs & %d sinks)
           to simulation\n", num_srcs*2, num_srcs, num_srcs);
    for (int i = 0; i < num_srcs; i++) {
        sprintf(src_name, "src%d", i);
        sprintf(sink_name, "sink%d", i);
        sim.add(new src(src_name, new sim_port(sink_name), i*2,
                      SRC_OK));
        sim.add(new sink(sink_name, new sim_port(src_name), i*2+1,
                      SINK_OK));
    }

    printf("main() - sim.run();\n");
    sim.run();

    printf("main() - done, return 0;\n");
    return 0;
}
```

Figure 3. main.C – Top Level of feltest Simulation Control Program

The `src.C` and `sink.C` files housed the constructors and processing code for the source and sink entity threads that were created by the HASE++ library in response to `main`'s `sim.add()` calls. The `src::body()` method was the event generating code for a source entity thread. It had three functions:

- Instantiate a simulation event object with a `sim_event ev;` statement
- Read parameters specifying the number of events (`src_events`) to generate, the average simulated event processing time (`e_dly`) to be used for each event, and the variance (`src_var`) to apply to this average (if any)
- Generate the requested number of events

The `for()` loop that generated events had the following sequence:

- Schedule an event for the associated sink entity for some time in the future. Depending on the value of the `src_var` parameter, this could be either a fixed or a variable time increment. If `src_var` is 0, it would be the actual value passed in the `sim_schedule()` call. If `src_var` was greater than 0, it would be a normally distributed variate with a mean of the value passed in the call and a variance of `src_var`. If this was the first event for this thread, it would be 0.0 (which indicates an immediate event).
- Wait for an “event completed” event to be returned from the sink entity
- Simulate some event processing for the requested time period (i.e. consume the number of CPU cycles necessary to pass this much time on a dedicated CPU)

Figure 4 lists the source code for the `src.h` and `src.C` programs:

```
// src.h - source entity header

#include "simpp.h"
#include "expt.h"
#include <stdio.h>

enum src_state { SRC_OK, SRC_BLOCKED };

class src : public sim_entity {
    sim_port out;
    int index;
    src_state state;
public:
    src(char*n, sim_port *out_i, int index_i, src_state state_i );
    void body();
};

// src.C - source entity code

#include "src.h"
#include <unistd.h>
#include <math.h>
```

```

#include <time.h>
#include <simstats.h>

src::src(char*n,sim_port *out_i, int index_i, src_state state_i) :
    sim_entity(n), out(*out_i), index(index_i), state(state_i)
{
    join_port(out, "out");
}

void src::body()
{
    char *entity_name;
    entity_name = get_name();
    char set_num = *(entity_name+3);

    sim_event ev;

    int src_events = get_int_param("src_events");
    int src_mean = get_int_param("e_dly");
    int src_var = get_int_param("src_var");

    sim_normal_obj delay("Delay Object", src_mean, src_var, index);
    sim_time delay_t = 1.0;
    sim_time work = 6.347210;
    work = work * (double)src_mean;
    long j = 0;
    long num_iterations = (long)work;
    double x = 0.0;
    for (int i = 0; i < src_events; i++) {
        if (src_var > 1) {
            delay_t = delay.sample();
        } else {
            delay_t = 1.0;
        }
        if (i == 0) {
            sim_schedule(out, 0.0, 0);
        } else {
            sim_schedule(out, delay_t, 0);
        }
        state = SRC_BLOCKED;
        sim_wait(ev);
        state = SRC_OK;
        if (src_mean > 0) {
            for (j = 0; j < num_iterations; j++) {
                x = sqrt((double)(j+1));
            }
        }
    }
}

```

Figure 4. src.h and src.C - Source Entity Thread Code

The `sink::body()` method was the event processing code for a sink entity thread. Its three functions were similar to `src::body()`, but complementary:

- Instantiate a simulation event object with a `sim_event ev;` statement
- Get the `e_dly` that was read-in by `src.body()`
- Process events generated by `src::body()`

The `for()` loop that responded to source events was similar to the `for()` loop in `src::body()`. It had the following sequence:

- Wait for an event from the source entity
- Simulate some event processing for the requested time period
- Schedule an “event completed” event for the associated source entity

Figure 5 lists the source code for the `sink.h` and `sink.C` programs:

```
// sink.h - sink entity header

#include "simpp.h"
#include "expt.h"
#include <stdio.h>

enum sink_state { SINK_BLOCKED, SINK_OK };

class sink : public sim_entity {
    sim_port in;
    int index;
    sink_state state;
public:
    sink(char*n, sim_port *in_i, int index_i, sink_state state_i );
    void body();
};

// sink.C - sink entity code

#include "sink.h"
#include <unistd.h>
#include <math.h>
#include <time.h>
#include <simstats.h>

sink::sink(char*n, sim_port *in_i, int index_i, sink_state state_i) :
    sim_entity(n), in(*in_i), index(index_i), state(state_i)
{
    join_port(in, "in");
}

void sink::body()
{
    char *entity_name;
    entity_name = get_name();
    char set_num = *(entity_name+4);
```

```

sim_time sink_delay = (sim_time) get_int_param("e_dly");
sim_event ev;
int i = 0;
long j = 0;
sim_time work = 6.347210;
work = work * (double)sink_delay;
long num_iterations = (long)work;
double x = 0.0;
while(1) {
    i++;
    sim_wait(ev);
    if (sink_delay > 0) {
        for (j = 0; j < num_iterations; j++) {
            x = sqrt((double)(j+1));
        }
    }
    sim_schedule(in, 1.0, 1);
}
}

```

Figure 5. sink.h and sink.C - Sink Entity Thread Code

In addition to the `feltest` simulation prototype, a basic thread test program, called `threadtest`, was developed. This test program needed to determine the best-case performance of threads on multiple CPUs. The performance of `feltest` could then be compared to this ideal performance.

Figure 6 lists the source code for the `threadtest` program:

```

// threadtest.c: arg[1]=num_cpus,arg[2]=num_thrds,arg[3]=num_usecs

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <math.h>
#include <pthread.h>

long num_usecs, num_iterations;

void *threadxx(void *arg) {
    long i;
    double one_usec = 5.3;
    double x, work;
    work = (double)num_usecs * one_usec;
    num_iterations = (long)work;
    for (I = 0; I < 5000; I++) {
        for (j = 0; j < num_iterations; j++) {
            x = sqrt((double)(j+1));
        }
    }
    return arg;
}

```

```

}

int main(int argc, char *argv[]) {
    int num_cpus, num_thrds, i, rc;
    int thrd_arg = 0;
    pthread_t tidp;

    if (argc != 4) {
        printf("Usage: threadtest num_cpus num_thrds num_secs\n");
        return 1;
    }

    num_cpus = atoi(argv[1]);
    num_thrds = atoi(argv[2]);
    num_usecs = atol(argv[3]);

    printf("\nthreadtest num_cpus=%d, num_thrds=%d, num_usecs=%d\n",
        num_cpus, num_thrds, num_usecs);

    rc = pthread_setconcurrency(num_cpus);
    if (rc != 0) {
        printf("pthread_setconcurrency() rc = %d\n", rc);
        return 1;
    }

    for (i = 0; i < num_thrds; i++) {
        rc = pthread_create(&tidp, NULL, threadxx, (void*)thrd_arg);
        if(rc != 0) {
            printf("pthread_create() error %d\n", rc);
            return 1;
        }
    }

    pthread_exit((void *)&rc);
}

```

Figure 6. threadtest - Basic Thread Test Program

threadtest was designed to allow the exploration of best-case threading where perfect, or almost perfect, speedup was achieved. This level of speedup is attainable when there is little communication or synchronization among threads or the main program, and the threads have a long execution time.

threadtest was controlled with command line parameters. The first parameter specified how many CPUs were to be used and the second parameter specified how many threads were to be run. While specifying more CPUs than threads was legal, the number of CPUs actually used was automatically reduced by the pthread library to the number of threads running. The opposite was not true. Specifying fewer CPUs than there were threads resulted in the threads sharing the available CPUs. Finally, the third parameter

specified the base number of microseconds each thread was to run before terminating. This value was then multiplied by a repetition factor (5,000 in the final version) to yield the actual run time for each thread. This approach allowed thread loop times as short as 500 microseconds or as long as many seconds to be specified.

The operation of `threadtest` was as follows. The main program issued a `pthread_setconcurrency()` call to reserve the requested number of CPUs. It then issued a `pthread_create()` call for each requested thread to create it and start it running. It finally issued a `pthread_exit()` call to wait for all the threads to terminate before terminating itself.

The thread compute loop was very machine and architecture dependent. Its purpose was to use CPU cycles – thus emulating some computational activity by a thread such as event processing. By knowing that a certain number of iterations through the loop would use a certain amount of CPU time on a single CPU, one could then set up thread workloads of varying duration. The number of iterations per second (5,300,000) was determined through testing. A number of runs were made using different values to create thread workloads. With thread workloads of 20 seconds, 16 and 32 threads, and 1 CPU used, the elapsed times would be expected to be 320 and 640 seconds, respectively. With the above input parameters, the elapsed times were within several hundredths of a second of the expected times. This was an accuracy of .00003%.

It followed from the above that the loop that comprised an iteration took approximately $1/5300000$ seconds to execute on an Origin 2000 with 250MHz CPUs. If 300MHz CPUs were used (such as at ARL), the loop would need to be recalibrated since it would be faster by 20% ($(300-250)/250$). Also, if the input parameter to `sqrt()` was changed to a constant, the loop would also have to be recalibrated because the C++ compiler would optimize this sequence. A test showed a reduction in CPU time of about 20% as a result of using a constant instead of a variable that was changing.

`threadtest` facilitated the exploration of CPU and thread interaction over a wide spectrum. In the baseline, scenario only one CPU was used. For example, if the run parameters specified 1 CPU, 32 threads, and 10 seconds of thread loop time (2,000 microsecond base loop specification), one would expect that the elapsed time would be 320 seconds (32 threads times 10 seconds each divided by 1 CPU). This was the same as what the sequential operation time would be if there were no threads. If this scenario were run with 2 CPUs, one would expect the elapsed time to be about half or 160 seconds. In the best-case scenario, 32 CPUs would be used. This would yield an elapsed time of 10 seconds (32 threads times 10 seconds each divided by 32 CPUs). This would represent perfect speedup.

For all practical purposes, perfect speedup in the context of a multithreaded simulation using a single FEL was not feasible. This was because the entity threads were not totally

independent, but tied to the single FEL. In this implementation, the FEL management code executed as part of the original thread started when the process was created. The FEL management code could be part of each entity thread but it was not apparent that there would be any advantage to this. The shared resource here was the FEL structure. As such, each entity must have had exclusive access to it in order to add or remove events from it. There was no potential for overlap while an event addition or removal was in process.

The initial system development was done remotely on a 128 CPU, 300MHz Origin 2000 system running IRIX 6.5 at the Army Research Lab (ARL) at Aberdeen Proving Grounds in Maryland. The final timing runs were performed on a 64 CPU, 250MHz Origin 2000 system running IRIX 6.5 at the Joint National Test Facility (JNTF) at Schriever AFB in Colorado Springs, Colorado. Access to this system was through a LAN-based SGI O₂ command line and/or X-Windows interface. This environment allowed the use of several debugging and testing tools that were not practical to use in the ARL environment, especially over a modem. One of the most useful of these was the “gr_osview” program. It displayed a graphical representation of CPU usage, updated four times a second, for all 64 CPUs on the system. It distinguished between system CPU time and user CPU time with color-coded, stacked, bar charts. This allowed the user to visually observe CPU loads. For example, if the `threadtest` program was started with 16 CPUs, the user could immediately see 16 CPUs jump to 100% user CPU busy. If this didn’t happen, then the user knew immediately that something was wrong. Another useful tool was the “top” program. `top` showed textually what, and where, processing threads were running at any time for a specific user. Its output was similar to the output of the Unix Process Status (`ps`) command. However, unlike `ps` which generates a single snapshot display and then ends, `top` runs continuously once started – updating the screen every second.

`feltest`’s operation was controlled by the `input.params` parameter file. The following shows the contents of a typical file:

```
16 (num_cpus)
32 (num_srcs)
1000 (src_events)
0 (src_var)
500 (e_dly)
100 (f_dly)
```

The meanings of the six entries in this file were as follows:

- (`num_cpus`) – the number of CPUs requested
- (`num_srcs`) – the number of entities that will generate events. In this study, a sink entity is also created for each source entity.
- (`src_events`) – number of events each source entity will generate during the course of the run.

- (`src_var`) – variance in time allowed between event generations. A variance of 0 generated a fixed time increment causing all future events to be added at the end of the FEL. Using 0 variance had the benefit of reducing FEL add-event overhead to a fixed value representing the time it took to add an event to the end of the FEL. This was also very close to the time it took to retrieve the next event from the front of the FEL. With this fixed overhead in place, the (`f_dly`) parameter could be used to more precisely study the effects of various amounts of FEL add-event overhead.
- (`e_dly`) – amount of time, in microseconds, that each entity thread should spend in its loop before scheduling an “event processed” event for the originator of the event.
- (`f_dly`) – amount of time, in microseconds, to be used to simulate FEL add-event processing overhead.

Based on the above `input.params` file, `feltest` would be run with the following parameters:

- 1 CPU
- 32 threads (16 sources and 16 sinks)
- 1000 events generated by each source (total events in simulation = 32,000)
- Fixed event time increments
- 500 microsecond simulated event processing time by each thread
- 10 microsecond simulated FEL add-event processing overhead

While the flexibility of the `input.params` file made it easy to construct different scenarios, each scenario was run only once. Several runs were usually desirable in order to observe best performance or to build statistics to describe the typical profile of a scenario’s operation. This was because, even when the system appears idle, there were still many processes and daemons alive in the system. These interfered with the operation of the test program causing variations in many of the captured data items such as % CPU Busy, Elapsed Time, System CPU Time, and Context Switches. Additionally, many runs with many different scenarios were necessary in order to be able to understand where performance gains are highest and where bottlenecks occurred.

A parametric study capability was implemented that allowed many scenarios to be constructed dynamically. It accomplished this by varying some of the parameters over a specified range (different for each parameter) and automatically ran each unique scenario. The following `exp.perl` Perl script provided this capability.

```
#!/bin/perl

# Version which varies N variables
# Gets range from range.src
@var_name;
@var_nameb;
@var_min;
@var_max;
```

```

@var_step;
@var_val;
$nvars = 0;

open (RANGEFILE, "range.src");

$runcmd = <RANGEFILE>;

$_ = <RANGEFILE>;
chop;          #
$outdir = $_;

while (<RANGEFILE>) {
    split;
    $var_name[$nvars] = $_[0];
    $var_nameb[$nvars] = '('.$_[0].')';
    $var_min [$nvars] = $_[1];
    $var_max [$nvars] = $_[2];
    $var_step[$nvars] = $_[3];
    $var_val[$nvars] = $var_min[$nvars];
    $nvars ++;
}

system( "cp input.params input.params.tmp" );

# For each variable...
$finished = 0;
while ($finished == 0) {
    # Run the routine with the given variables.
    # Replace param vals with $vals
    open (NEWPARAMSFILE, ">input.params") || die "Can't open o/p\n";
    open (PARAMSFILE, "input.params.tmp");
    while (<PARAMSFILE>) {
        split;
        $found = -1;
        for ($j=0; $j<$nvars && $found== -1; $j++) {
            if ($var_nameb[$j] eq $_[1]) {
                $found = $j;
            }
        }
        if ($found != -1) {
            printf NEWPARAMSFILE "$var_val[$found]
                                $var_nameb[$found]\n";
        } else {
            printf NEWPARAMSFILE "$_";
        }
    }
    close (NEWPARAMSFILE);
    close (PARAMSFILE);
    system("cat input.params");

    for ($j=0; $j<5; $j++) {
        system ($runcmd);
    }
}

```

```

# Store results for future processing.
$resdir = $outdir;
for ($j=0; $j<$nvars; $j++) {
    $resdir = $resdir."/".$svar_val[$j];
}
printf $resdir."\n";
system ("mkdir -p $resdir");
system ("cp -p tracefile $resdir");

# Increment variables
$done = 0;
for ($j=$nvars-1; $j>=0 && $done==0; $j--) {
    # Try to increment variable indicated by $j
    if ($var_step[$j] > 0) {
        $var_val[$j] += $var_step[$j];
    } else {
        $var_val[$j] *= 10;
    }
    $done = 1;
    if ($var_val[$j] > $var_max[$j]) {
        $var_val[$j] = $var_min[$j];
        $done = 0;
        if ($j == 0) {
            $finished = 1;
        }
    }
}
}

system("cp range.src input.params $outdir");

```

Figure 7. Example `exp.perl` Perl Script File

The basic logic flow of this script was as follows. First it read the `input.params` file. It then read the `range.src` file. This file specified which parameters from the `input.params` file were to be varied, over what ranges, and by what increments. The following shows an example `range.src` file.

```

/usr/bin/time -l feltest
results
e_dly 100 1000 100
f_dly 0 100 50

```

There was an order to the lines in the `range.src` file. The first line was the program execution statement; it caused the `feltest` program to execute. The second line specified the subdirectory where test results for each scenario were to be written. The remaining lines (in this case 3 and 4) specified the parameters in the `input.params` file to be varied, the starting value, the ending value, and the increment. In this example, `f_dly` was varied from 0 to 100 in increments of 50 and `e_dly` was varied from 100 to

1000 in increments of 100. Sometimes, a very wide range of parameter values was desired (for example, in the initial bracketing of parameter sensitivity). To assist in these situations, an increment of 0 could be specified. This caused the associated parameter to be increased by an order of magnitude (i.e. multiplied by 10) for each iteration.

With even just the two parameters being varied, a surprising number of runs resulted. The example above would cause 30 unique scenarios to be run. And, as implemented in the `exp.perl` file, each scenario would be repeated 5 times. Therefore, this `range.src` file would generate a total of 150 runs of the `feltest` program. Note that this would only be applicable to one “number of CPUs” submittal. If the `range.src` file included the following statement:

```
num_cpus 2 16 2
```

8 different CPU complements (2, 4, 6, 8, 10, 12, 14, and 16) would be run. This would result in a total of 1,200 `feltest` runs. If the average run time was 30 seconds, this would total 10 hours.

The basic timing problem was that the `feltest` was not the only process running in the system. As previously mentioned, even when the system appeared to be idle, there were still many processes and daemons active. The system’s CPUs, caches, memory, operating system (IRIX) facilities, disks, etc., were shared with these other programs. The main measurement of improvement for the `feltest` program when CPUs were added was elapsed time. However, elapsed time could be adversely affected (i.e., increased) by other activity in the system. So, as described in the parametric study section, a number of runs were required for each scenario in order to ensure that one or two with minimal interference were run. Additionally, other measurements were affected by system activity (e.g., % CPU busy). An additional goal of this research with the `feltest` program was to quantify some of these relationships.

The system time command works as follows. The program to be timed is specified as the last parameter of the `/usr/bin/time -l` command. The “-l” option requests the long, resource usage, report format. This report is written to `stderr`. Figure 8 shows an example of this output after an `feltest` run.

```
elapsed time ..... 3.840
  user CPU ..... 3.469
  system CPU ..... 0.329
  percent busy ..... 98.9%%
page faults ..... 175
  page reclaims ..... 175
  page ins ..... 0
context switches and swaps ..... 18
  voluntary context switches ..... 17
  involuntary context switches ..... 1
```

```

swaps ..... 0
Number of I/O operations ..... 3
  block input operations ..... 1
  block output operations ..... 2
signals received ..... 0

```

Figure 8. Example “/usr/bin/time -l ...” Command Output

Following are the definitions of the time and resource usage data items collected and displayed in the `stderr` output:

- `elapsed time` – total (wall-clock) time a program took to execute in seconds and thousandths of seconds
- `user CPU` – time during a program’s execution when it was executing in “user” mode (also referred to as “process virtual time”). It is presented in seconds and thousandths of seconds. It includes the execution of library routines in user mode. Libraries do their processing in user mode so their execution time is charged to the process that calls their routines. However, some of these libraries make system calls on behalf of the calling process. The resulting time the system spends executing calls from the library on behalf of the process is charged to the system CPU time that is accumulated against the process. Finally, it is important to note that when multiple CPUs are used by a process, as is possible with `pthread`s, the reported user CPU time will usually exceed `elapsed time` (potentially by a large amount when many CPUs are used). This occurs because user CPU time remains virtually constant regardless of the number of CPUs used while `elapsed time` usually decreases as additional CPUs are used.
- `system CPU` – time during a program’s execution when the system was executing code on behalf of the program
- `percent busy` – CPU utilization during the indicated `elapsed time`. It is calculated as: $((\text{user CPU} + \text{system CPU}) / \text{elapsed time}) * 100$. It is displayed to an accuracy of 1 decimal place (i.e. `dd.d%`). Since, ideally, `elapsed time` should go down as CPUs are added, `percent busy` for multiple CPU runs should exceed 100%.
- `page faults` – total number of page faults the process experienced. A high page fault rate is an indication of a memory bound situation. It is the sum of `page reclaims` and `page ins`.
- `page reclaims` – number of page faults that resulted in a page being reclaimed from the page cache.
- `page ins` – number of page faults that resulted in a page being read from disk.
- `context switches and swaps` – total of voluntary context switches, involuntary context switches, and process swaps.
- `voluntary context switches` – number of voluntary context switches. These result from explicit yields of the CPU via calls to system support functions

such as `sleep()`, `nanosleep()`, `sched_yield()`, `sginap()`, and I/O calls as well as from contention on resources. A high number of voluntary context switches and large amounts of idle time often indicate a CPU resource problem.

- `involuntary context switches` – number of involuntary context switches. These result from a process being switched out at the end of its CPU time slice or being preempted by a higher priority process.
- `swaps` – number of times the process was swapped to secondary (usually disk) storage. Non-zero values indicate that the system's memory was oversubscribed.
- `number of I/O operations` – total number of block input operations and block output operations performed for the process. Block I/O operations read and write blocks of data from/to disk.
- `block input operations` – number of disk input operations that were performed for the process.
- `block output operations` – number of disk output operations that were performed for the process.
- `signals received` – number of signals received by the process.

For the `threadtest` program, only the number of CPUs used varied between runs. Therefore, the complexity of the `input.params` and `exp.perl` mechanisms were not necessary. A simple shell script, such as the following, yielded the desired results.

```
/usr/bin/time -l threadtest 1 32 500
/usr/bin/time -l threadtest 2 32 500
/usr/bin/time -l threadtest 4 32 500
/usr/bin/time -l threadtest 6 32 500
/usr/bin/time -l threadtest 8 32 500
```

This script would cause `threadtest` to run 5 times – each run having 32 threads and allocating 500 microseconds of CPU time per thread. The first run uses only 1 CPU, while the following runs use 2, 4, 6, and finally 8 CPUs. If multiple runs per configuration were desired for each CPU complement, each command line could just be repeated in the script file. Elapsed times from runs using single CPUs have typically experienced minimal variations (generally less than .05%). Multiple CPU runs have tended to vary more widely. The system `stdout` and `stderr` output streams containing the measurements from runs were piped to disk files with standard piping extensions. C++ data reduction programs were used to read these files and create Comma Separated Value (CSV) files for later input to Microsoft Excel spreadsheets. For the `feltest` program, both the `stdout` and the `stderr` files were used because the `stdout` file contained information indicating what parameters were used for each run.

Results

Plan on probably putting the 3 graphs from my defense here along with a fairly short summary. Would have put the graphs here now, but I'm having placement problems. There's some option I have to turn off to get them where I want them and sized correctly.

Discussion

Not sure what to put here. I'm thinking some of the explanation in the Methodology section should be moved into this section.

Mark H. Butler has over 38 years in the computer industry, the last 4 with TRW Systems in Colorado Springs. Mark has worked with several major computer HW/SW companies in the past and was also with TRW in Los Angeles for several years in the early 80s. Mark is currently a Senior Software Engineer assigned to the Joint National Test Facility in Colorado Springs where his work includes the optimization of Parallel Discrete Event Simulation (PDES) and alternative platforms for High Performance Computing (HPC) including Beowulf cluster computing. Mark holds a Bachelor of Science degree in Management and Finance from Golden Gate University in San Francisco. He also holds a Master of Science degree in Computer Science and a Doctor of Science degree in Computer Science (D.CS.) from Colorado Technical University in Colorado Springs, Colorado, where he serves as an Adjunct Professor of Computer Science.

Bo I. Sandén is a Professor of Computer Science at Colorado Technical University in Colorado Springs, Colorado, where he teaches software engineering, simulation and modeling. His main research interest is software design using multi-threading, and he has published multiple papers on that topic. He is also the author of a book on software construction. Before entering academia, he spent 15 years as a software designer and project lead with UNIVAC and Philips. Bo holds a Master of Science in Engineering Physics from the Lund Institute of Technology, Lund, Sweden, and a Ph.D. in Computer Science from the Royal Institute of Technology, Stockholm, Sweden.