SPLASH-2 Ocean Performance Analysis

Purpose

This paper discusses the performance of the pthread version of the SPLASH-2 Ocean simulator described in "The Translation of SPLASH-2 Ocean Simulation to Pthread and MPI". Performance will be described both in terms of actual time required and speedup. In addition the performance of the pthread program will be compared to that of the same Ocean simulation using an SGI specific translation of the PARMACS library.

This paper is intended to be support some of the results described in the "The Translation of SPLASH-2 Ocean Simulation to Pthread and MPI". It is not intended to stand alone as it has little value to someone not familiar with the previous paper.

About the Pthread Program

The techniques used in the pthread version of the Ocean simulation are discussed in my earlier paper "The Translation of SPLASH-2 Ocean Simulation to Pthread and MPI".

About the SGI Specific Program

The SGI specific version of the Ocean simulation was built using a distribution of the ANL PARMACS library which uses traditional Unix and SGI native commands. No attempt on my part was made optimize the PARMACS implementation. The only modifications that I made were to compile using the –n32 option, and to double the amount of shared memory allocated in order to obtain results for the 514×514 ocean simulation with 16 processes.

The Ocean source used for the SGI specific implementation is nearly identical to the source code that the pthread implementation was built from, and both versions of the program use the same PARMACS primitives: MAIN_ENV, MAIN_INITENV, MAIN_END, EXTERN_ENV, CREATE, WAIT_FOR_END, LOCKDEC, LOCKINIT, LOCK, UNLOCK, BARDEC, BARINIT, BARRIER, G_MALLOC, and CLOCK.

For this implementation, CREATE forks a new process, instead of using threads, so all values are copied at the time of the fork, this means that the created process starts out with a copy of all the process specific address space, but it is only a copy. After the fork, one process cannot effect the processes specific memory of another process. Consequently memory allocations on the heap, allocated by malloc(), cannot be share between processes. In order to provide for a memory space that may be shared between processes, G_MALLOC is implemented using System V shared memory and memory arenas similar to my implementation of shared memory under MPI.

The SGI implementation of the Ocean simulation uses the Irix specific uscsetlock, usunsetlock, and barrier functions from the ulocks library to implement LOCK, UNLOCK, and BARRIER.

Data Collected

For this performance analysis, I collected four different sets of data:

- Pthread program with a 258×258 ocean
- Pthread program with a 514×514 ocean
- Forking SGI program with a 258×258 ocean
- Forking SGI program with a 514×514 ocean

For the pthread data sets, I collected data for 1, 2, 4, 8, 16, 32, and 64 processes, where the function pthread_getconcurrency indicated a concurrency level equal to the number of processes (threads) created. The forking SGI program only ran to completion for 1, 2, 4, 8, and 16 processes, so there is no data for 32 and 64 process in the SGI data sets. The 32 and 64 process versions reported that utime was exceeded by each process.

All data collections were obtained from the same SGI Origin 2000 (modi4), running Irix 6.5. The profiling tool perfex was also used to obtain additional performance information.

Each data set contains three executions of their corresponding Ocean program from the command line using the command:

```
% perfex –a –y OCEAN –p x –n y
```

where *x* is the number of processes used and *y* is the ocean dimension.

Results

Execution Time

Figure 1 graphically depicts the average execution time for the data collected for a 258×258 ocean. The x-axis is number of processes, and the y-axis is the average execution time for each of the three executions in μs. Figure 2 graphically depicts the same for the 514×514 ocean problem. For each execution there are two separate time values provided: time with initialization, and time without initialization. In general, the initialization is done before the problem solving processes are created, so time with initialization provides for analysis of total problem solving time, while time without initialization is the time that Ocean uses parallel processes, and may be used to better reflect the speedup due to parallelization.

From the graphs, it can be seen that in most cases, the difference between execution time of the pthread program and the fork program is on the order of milliseconds. The interesting observation that can be obtained from the graphs is that for most cases the pthread problem ran faster than the fork program on the 258×258 ocean, but the opposite is true for the 514×514 ocean. When I compared the perfex results to come up with an explanation, I could not come up with one. Regardless of problem size, the fork version executed more instruction cycles, and both versions had about the same amount of cache misses. The only explanation that can come up with is that the fork version of Ocean is worse at handling communication and process synchronization.

As stated in the previous paper, the communication to computation ratio for the Ocean simulation is sqr(p)/n, where p is the number of processes and n is the ocean dimension. The case where the fork version of Ocean performed poorer than the pthread version had higher communication to computation ratios than the cases where it performed better.

Poor performance handling the communication might also explain why the fork version of Ocean is unable to complete without exceeding utime limit for the 32 and 64 process case. These case have more communication overhead, thus increasing the time required for synchronization. Further testing using lock an barrier benchmarking programs might be useful in measuring the performance of the techniques used by each version of the ocean simulation.

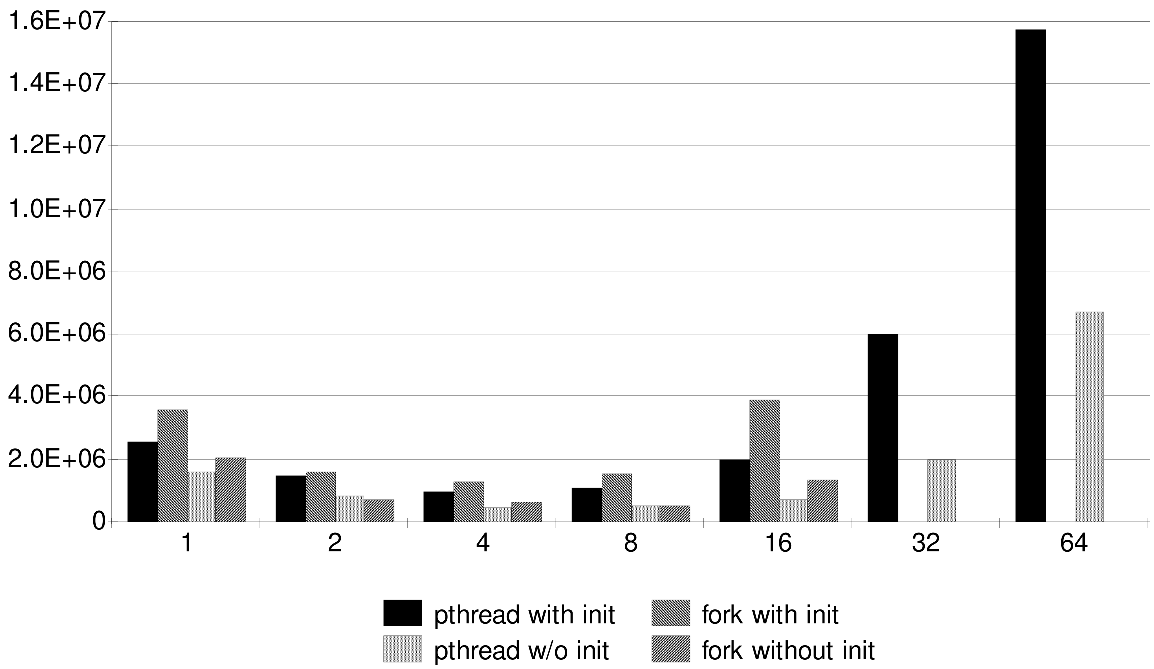

Figure 1 Execution Time for 258 x 258 Ocean

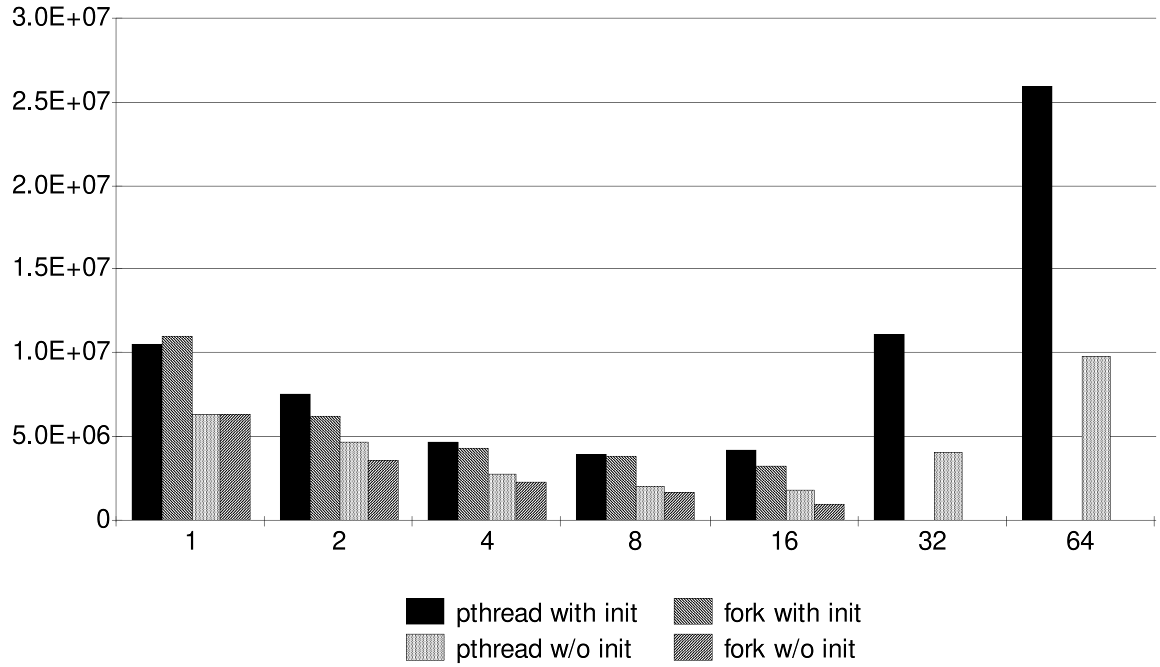

Figure 2 Execution Time for 514 x 514 Ocean

Speedup

The problem constrained speedup for a program may be measured by the equation speedup(p) = time(1)/time(p). The tables that follow provide the problem constrained speedup calculations for each program execution in my data set. Speedup values are provided for executions both including and not including initialization. The times with initialization provide a measure of the application speedup, while the times without initialization provide a speedup measure for just the parallelized portion of the execution.

In every case the speedup without initialization was greater than the speedup with initialization. I'm not sure that any useful statements may be made comparing the speedup of the pthread program to that of the fork program, but the data for both programs shows that after the number of processes reaches four, little to no speedup is gained by increasing the number processes. In fact every problem sets, speedup will start to decrease at 8 or 16 processes.

As with execution time, I believe that increased communication costs must be the reason for speedup degradation. There are only two costs that are incurred when processes are added, initialization cost and communication cost. Since the same trend in decreased speedup is seen with and without initialization, the added communication overhead is likely to be the reason for speedup decreasing after a certain amount of processes are added.

Number of Processes	Average Execution Time (μs)	Speedup
1	2540328	1
2	1441426	1.762371429
4	967680	2.625173611
8	1105911.333	2.297044911
16	2003882.667	1.267702966
32	6017423.333	0.422162088
64	15728558.67	0.16151054

Table 1 Pthread 258 x 258 Simulation Time Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	1574559.667	1
2	841634.6667	1.870835089
4	452256.6667	3.481562092
8	489045.3333	3.219659936
16	708292.6667	2.223035393
32	1993176	0.789975229
64	6706672	0.234775112

Table 2 Pthread 258 x 258 Simulation Time not Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	10517738	1
2	7545647.667	1.393881409
4	468259	2.246133982
8	4001743	2.628289223
16	4217530.667	2.493814232
32	11153521.33	0.942997076
64	25880801.67	0.406391507

Table 3 Pthread 514 x 514 Simulation Time Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	6359154.667	1
2	4709130.333	1.350388334
4	2799802.667	2.271286738
8	2070585	3.07118745
16	1780454	3.571647831
32	4081778.333	1.557937288
64	9769122	0.650944339

Table 4 Pthread 514 x 514 Simulation Time not Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	3582254.667	1
2	1572100.333	2.278642521
4	1292162	2.772295321
8	1519581.667	2.357395292
16	3900624	0.918379897

Table 5 Fork 258 x 258 Simulation Time Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	2055371.333	1
2	686522.6667	2.993887068
4	628537.3333	3.270086317
8	498038.3333	4.126934004
16	1354982.333	1.51689899

Table 6 Fork 258 x 258 Simulation Time not Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	10965984.67	1
2	6222961	1.762181165
4	4262976.333	2.572377562
8	3805219	2.881827476
16	3250923	3.373191142

Table 7 Fork 514 x 514 Simulation Time Including Initialization

Number of Processes	Average Execution Time (μs)	Speedup
1	6370870	1
2	3606863.333	1.766318657
4	2315217	2.751737742
8	1664643.667	3.827167416
16	952475.6667	6.688748304

Table 8 Fork 514 x 514 Simulation Time not Including Initialization

Conclusion

For each program increased communication requirements lead to degradation of performance. For case which have sufficiently low interprocess communication, speedups approaching 50 to 70% may be readily achieved. In case where the communication to computation ratio is low the fork version of the Ocean simulation performs better than the pthread version. However, in each data set there is a point where the communication becomes larger and the fork version of the ocean simulation either experiences a greater slowdown, or the fails to complete before the utime limit is exceeded.