The Translation
of
SPLASH-2 Ocean Simulation
to
Pthread and MPI

by

Michael Dipperstein
mdippers@cs.ucsb.edu
CS240 Fall 1999

Table of Contents

List of Figures

1 Description

This paper discusses the lessons learned and the results of an attempt to translate the Stanford Parallel Applications for Shared Memory (SPLASH-2) ocean simulation program (Ocean) to both Pthread an MPI implementations. Ocean is one of several applications in the SPLASH-2 library, intended to benchmark performance on various systems. Initially I set out to use the Ocean program to compare Pthread and MPI performance on both an SGI Origin 2000 and a Cray T3E. Though I did not accomplish my goals, there are still several valuable lessons learned to report along the way.

1.1 Problem Overview

Ocean is a C program that calculates a simulated ocean's currents over a cubical region. The calculations are performed in parallel using the Argon National Laboratory's (ANL) PARMACS macros to obtain parallelization (see section 1.3).

Ocean makes its current calculations at multiple horizontal layers of an ocean region. The current at each layer is influenced by surface wind, floor pressure, and eddies, all of which are constant conditions within the application, and cannot be change form one iteration to the next.

The current simulation is expressed as a boundary value problem in which the ocean area is divided into a grid of ocean layers. Streamfunctions between layers effecting each layer.

Because the ocean is divided into a grid, a grid solver may be used to solve the problem. The SPLASH-2 version of Ocean uses the Red-Black Gauss-Seidel grid solving algorithm. The program terminates when difference between iterations settles within a user provided tolerance. Varying the tolerance may change the output of the program.

In addition to tolerance, number of processes, grid size, distance between grid points, and time steps may also be modified at start of execution and change the results of the program. To assure correctness of the translations, a copy of correct outputs (correct.out) using the default parameters is provided with the program.

1.2 Problem Partitioning

The ocean section used in the simulation is represented as an $N \times N$ grid,where N is a power of 2 (+2 for boundary conditions). Processors are also arranged in a logical grid, and assigned a subgrid section of the ocean to perform current calculations. Figure 1 depicts the allocation of 16 processors to subgrid sections for an 18×18 ocean. The dashed lines represent grid cells and the dark lines represent subgrid boundaries.

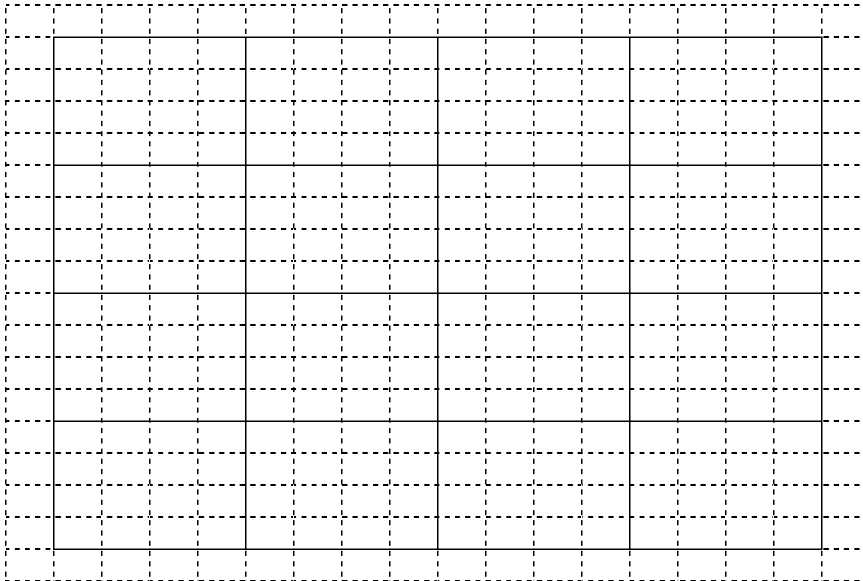

Figure 1 Subgrid Partitioning

In order to support the program's calculations, processes are required to share boarder data with each other. Since processes are arranged in logical grids of subgrids, the communication between processes increases proportionally with the square root of the number of processes.

As previously stated, the simulated ocean is divided into layers. The minimum number of ocean layers is 5 (2 + top, bottom, and middle layers). Currents at different layers are represented by a matrix, Ψ. Solving for Ψ at each layer is done in a 10 phase process regardless of the number of levels. Phases are delimited with a *BARRIER* instruction to allow for synchronization and communication between processes. Since a grid solver is used, it is only required that a process remain in the barrier until all it's neighbors have entered the barrier, however this is an optimization over the existing simulation which uses traditional barrier semantics requiring all processes to enter the barrier before any may leave.

Figure 2 illustrates the phase partitioning of each layer. Each row in Figure 2 is representative of a phase while the columns are used to indicate computational dependencies. This figure makes it clear that further optimizations may be performed allowing part of one phase to begin provided all the computations it depends on in a previous phase have been completed. In many cases this may allow a phase to begin earlier, though I am not sure if this will have any impact on the total processing time, since it is unclear to me whether or not this would allow an otherwise idle process to make progress.

<table>
<tr><td>Put Laplacian of Ψ_L in $W1_L$</td><td>Put Laplacian of Ψ_3 in $W1_3$</td><td>Copy Ψ_L, Ψ_3 to T_L, T_3</td><td>Put Ψ_L - Ψ_3 in W2</td><td>Put computed Ψ_2 vals in W2</td><td>Initialize γ_a and γ_b</td></tr>
<tr><td colspan="2">Add f values to columns of $W1_L$ and $W1_3$</td><td>Copy Ψ_{LM}, Ψ_{3M} into Ψ_L, Ψ_3</td><td></td><td></td><td>Put Lapacian of Ψ_{LM}, Ψ_{3M} in $W7_{L3}$</td></tr>
<tr><td colspan="2">Put Jacobians of ($W1_L$, $T1_L$), ($W1_3$, $T1_3$) in $W5_l$, $W5_3$</td><td>Copy T_L, T_3 into Ψ_{LM}, Ψ_{3M}</td><td></td><td></td><td>Put Lapacian of $W7_{L3}$ in $W4_{L3}$</td></tr>
<tr><td colspan="2"></td><td></td><td colspan="2">Put Jacobian of (W2, W3) in W6</td><td>Put Lapacian of $W4_{L3}$ in $W7_{L3}$</td></tr>
<tr><td colspan="6">Update the γ expressions</td></tr>
<tr><td colspan="6">Solve the equations for Ψ_a and put the result in γ_a</td></tr>
<tr><td colspan="6">Compute the integral of Ψ_a</td></tr>
<tr><td colspan="3">Compute $\Psi = \Psi_a + C(t)\Psi_b$ (Note: Ψ_a and now Ψ are maintained in the γ_a matrix)</td><td colspan="3">Solve the equation for Φ and put result in γ_b</td></tr>
<tr><td colspan="6">Use Ψ_a and Φ to update Ψ_L and Ψ_3</td></tr>
<tr><td colspan="6">Update streamfunction running sums determine when to stop running program</td></tr>
</table>

Note: Horizontal lines represent synchronization points among all processes, and vertical lines vertical lines spanning phases demarcate threads of dependence.

Figure 2 Phase Partitioning

1.3 Parallel Processing Primitives

The entire SPLASH-2 library was written using Argon National Laboratory's (ANL) PARMACS parallel processing macros for shared memory programs. The PARMACS macros are intentionally general, the idea being that macros may be translated into primitives of any other shared memory parallel processing scheme.

The template for the PARMACS macros utilizes the M4 macro language, however in the translations that I wrote, the C preprocessor was used, to avoid any errors that I might make due to my lack of familiarity with M4.

For my translation of the Ocean program to pthreads and MPI, I took an additional short cut. Instead of translating the entire set of PARMACS macros to pthreads and MPI, I only translated the macros used by Ocean.

The following PARMACS macros are used by the Ocean simulation:

Environment initialization

MAIN_ENV, MAIN_INITENV, MAIN_END, EXTERN_ENV

Process creation and termination:

CREATE, WAIT_FOR_END

Locking:

LOCKDEC, LOCKINIT, LOCK, UNLOCK

Barrier:

BARDEC, BARINIT, BARRIER

Memory allocation:

G_MALLOC

Time Keeping:
CLOCK

2 Challenges

2.1 General Challenges

The very first challenge I faced was understanding intentions of the Ocean program. Ocean is sparsely commented. The bulk of the comments are headers which state that Stanford copyrighted the program, but anyone can modify it, and Stanford accepts no liability. The lack of comments would not be that bad, if it wasn't compounded by the fact that much of the programs computations are performed on variables with names non-descriptive names like W, xtemp, and jm.

A challenge that I didn't realize until I had done most of the translations was that the Origin 2000 defaults to 64-bit alignment, but the Ocean code was written for machines with a 32-bit alignment. This caused segmentation faults and bus errors to occur for certain problem sizes. The solution was to compile the code using the –n32 switch. Realizing that Ocean deliberately used 32-bit alignment was the difficult part.

Another challenge was to write translations for macros that were not supported in pthreads and MPI. Both pthreads and MPI are missing different components defined in PARMACS. The rest of this section discusses challenges specific to each translation.

2.2 Pthread Challenges

The biggest challenges specific to the pthreads translation of the Ocean program are that CREATE only creates one process at a time and does not use thread handles, which WAIT_FOR_END needs so it can join all the threads that arrive. In my initial implementation of CREATE, I tracked the number of times CREATE was called. The first time a thread was created, I allocated memory for a size 1 array of thread pointers. Each successive CREATE call would realloc the array one bigger. Not only was realloc time consuming, but certain executions resulted in segmentation faults. My solution to this problem was to cheat. Once Ocean starts, the total number of threads it will CREATE is known. I pass that number to CREATE, so that a single allocation for an array of thread pointers can be done when CREATE is first called, and realloc is never called.

A more challenging problem to overcome was the fact that pthreads has no barrier functions. BARDEC, BARINIT, BARRIER had to be built from other operations available to pthread programs. The requirements and implementation of the barrier functions is discussed in paragraph 3.1.

2.3 MPI Challenges

The challenges of translating Ocean to MPI are much more difficult to overcome than the challenges of translating Ocean to pthreads. The following is a list of the most significant challenges of translating Ocean to MPI:

- MPI processes exist before CREATE is called
- MPI version 1.2 does not support shared memory
- MPI processes do not share data stored in global C variables
- MPI does not support locks

Since MPI processes exist before CREATE is called, the code which is executed before the CREATE will be executed on all processes. The following code fragment illustrates the classes of problems that arise from processes existing before the CREATE statement:

```
int i, j;
int *k;

main()
{
    printf("program started\n");
    k = (int *)G_MALLOC(sizeof(int));
    for(i = 1; i < 10; i++)
    {
        j += i;
    }

    CREATE(...)
          ⋮
}
```

When multiple processes are running, it is unclear whether or not they should all execute printf statement, however it is clear that they should all execute the G_MALLOC statement. It is also clear that only one process should execute the for loop. I could not come up with an obvious technique for automatically determining which code should be executed on by a single process and which code should be executed by all processes, so I manually partitioned the code using if (myRank == 0) statements.

Just as challenging of problem was developing a scheme for shared memory allocation. Ocean was written for the shared memory model, however, MPI 1.2 does not support the shared memory model. There are two approaches to this problem, one is to rewrite Ocean to use the SPMD model, the other is to develop a shared memory scheme that works under MPI. I chose the later, because I wanted to be able to compare the performance of MPI code with that of pthread code from a common source. The implementation of shared memory suited for use with MPI code is discussed in paragraph 4.1.

Another problem which resulted from MPI not using the shared memory model is that non-dynamic global variables that are intended to be shared by processes are not. I attempted to solve this problem redefining the global variables as pointers and using G_MALLOC to allocate space for them.

For every global variable of the form:

int foo;

I wrote the following code:

#define foo *g_foo
g_foo = (int *)G_MALLOC(sizeof(int));

A challenge that is only remotely related to MPI not supporting shared memory, is that MPI does not support locks (SPMD doesn’t need locks since it doesn't share resources). LOCKDEC, LOCKINIT, LOCK, UNLOCK had to be written using shared memory and primitives available to MPI. Since LOCK and UNLOCK require atomic operations and MPI doesn’t supply them, non-MPI operations were required. The primitives I chose are specific to the SGI Origin 2000, so the MPI code is not portable. The implementation of locks under MPI is discussed in paragraph 4.2.

3 Pthread Implementation

The only PARMACS macros used by Ocean that did not have relatively straightforward translations into pthread support code where the barrier macros. For most other macros, there's a near one-to-one correspondence with pthread functions. The files tlib.h and tlib.c implement Ocean's PARMACS macros for pthreads.

3.1 Barrier functions

The barrier macros used by Ocean are: BARDEC, BARINIT, and BARRIER. The implementation for BARRIER is discussed in the subsections which follow. The other two macros are straightforward to implement.

The barrier functions to support PARMACS have the following requirements:

- Processes that enter a barrier may only leave after all other processes that the barrier is set for have entered the same barrier.
- Multiple simultaneous barrier instances must be allowed for. That is the same barrier may be active with two different barrier counts as long as the sum of the counts does not exceed the total number of processes. Ocean does not use this feature.
- Barrier must be reentrant, allowing multiple processes to enter before others exit.
- Processes from the same barrier episode must be able to be in the barrier at the same time.
- Processes from different barrier episodes must be able to be in the barrier at the same time, without interfering with each other episodes. If one group of processes is leaving the barrier, another group may enter the barrier, and the group entering cannot cause any processes from the group leaving to get caught. Nor can processes from the group leaving cause an entering process to leave early.

3.1.1 BARRIER_TYPE

Variables of the type BARRIER_TYPE are used to keep track of the status of each barrier. A BARRIER_TYPE variable contains two counters, waiters and episode. Waiters is the number of processes in the barrier for the current episode. It is used to determine if all the processes that are suppose to enter the barrier have entered it. The episode counter tracks the current episode and prevents processes from one barrier episode from interfering with another. Only processes that have acquired the barrier's mutex may change either counter.

```
typedef struct
{
    unsigned int waiters;
    unsigned int episode;
    pthread_mutex_t mutex;
    pthread_cond_t cond;
} BARRIER_TYPE;
```

3.1.2 BARRIER

The Barrier function implements the PARMACS BARRIER macro. When a process enters Barrier, it first attempts to acquire the barrier lock. Once the lock is acquired, a process reads the episode counter and copies the value in a local variable. That value will be retained while the processes is in the barrier. If the value of the episode counter ever changes, it must be that case that the process can leave the barrier.

After the barrier episode is copied, the process will increment the waiters counter and determine if it is the last one required to enter the barrier. If the process is the last one required to enter the barrier, it increments the episode count, resets the waiters count, broadcasts an end of wait to all waiting processes, releases the mutex and leaves the barrier.

If the process is not the last process that is required to enter the barrier, it will conditionally wait until the episode number has changed and it has received a broadcast from the last process.

The following is the C source code for Barrier:

```
void Barrier(BARRIER_TYPE *barrier, int count)
{
    unsigned int thisEpisode;

    /* decrement barrier count under mutex protection */
    pthread_mutex_lock(&barrier->mutex);

    /* get the thread's current episode */
```

```
thisEpisode = barrier->episode;
++barrier->waiters;

if (barrier->waiters == count)
{
    /*************************************************************
    * Last thread, change episode for next iteration so that threads
    * that now enter this routine to block even if all threads from
    * the previous iteration have not left the routine.
    *************************************************************/
    ++barrier->episode;
    barrier->waiters = 0;
    pthread_mutex_unlock(&barrier->mutex);
    pthread_cond_broadcast(&barrier->cond);
}
else
{
    while (barrier->episode == thisEpisode)
    {
        /* wait for broadcast on current episode (not any other) */
        pthread_cond_wait(&barrier->cond, &barrier->mutex);
    }

    /* unlock mutex before leaving the routine */
    pthread_mutex_unlock(&barrier->mutex);
}
```

4 MPI Implementation

The only PARMACS macros used by Ocean that did not have relatively straightforward translations into MPI support code where the global memory allocation and lock macros. For most other macros, there is a near one-to-one correspondence with MPI functions. The files mlib.h and mlib.c implement Ocean's PARMACS macros for MPI.

In addition, the MPI version of Ocean that attempts to correct the utilization of global variables use definitions in gdefs.h to redefine all shared global variable accesses as dereferencing of a pointer to data allocated by G_MALLOC macro. The actual calls to G_MALLOC that allocate memory space for the global variables have been added to main.c.

4.1 Shared Memory

The only shared memory macro used by Ocean is G_MALLOC. The purpose of G_MALLOC is to dynamically allocate memory which is global to all active processes.

The functions which support G_MALLOC have the following requirements:

- Allocated space must be addressable by all participating processes.
- Allocated address must be the same for all processes.
- Standard variable operators and must work on G_MALLOCed variables.

4.1.1 Creating a Shared Memory Space

Though I have not found any documentation, I believe the second parameter in the MAIN_INITENV macro is intended to be the size of the shared memory space available. Following through with this belief, I wrote mlib.c so that it creates and initializes shared memory space as part of its expansion of MAIN_INITENV.

The global memory used by the MPI translation of Ocean is System V shared memory associated with a memory arena. In order to uses memory arenas, an additional constraint had to be added. Prior to the creation of a shared memory arena, the number of process which will have access to it must be known in advance. Since these routines were written for MPI code, where the number of processes is known, I didn't consider that to be a real limitation.

Memory is allocated by the shmget function, and then process 0 attaches it to an address. To ensure all other process share the memory space at the same address, the address is broadcast to all other processes, which then attach the shared memory to that address.

The shared memory is then given a file name so that it may be used by the arena. Process 0 then creates the memory arena, which all other process join. Finally process 0 is given a pointer to the arena, which it will use for memory allocation (see paragraph 4.1.2)

The following C code implements the shared memory creation with the ShmemInit function

```
void ShmemInit(size_t shmSize, int maxUsers)
{
    int rank;
    key_t shmKey;
    usptr_t *usPtr;

    /* Get process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    /* Get ID and create memory pool.  Only 1 PE needs to do this */
    if (rank == 0)
    {
        /* Generate reasonablly unique key */
        shmKey = (key_t)getpid();
        shmKey = (shmKey << 16) | shmKey;
        shmId = shmget(shmKey, shmSize, IPC_CREAT|SHM_R|SHM_W|444);

        if(shmId == -1)
        {
            ExitError(rank, 1, "shmget error");
        }
    }

    /* Broadcast shmId to all PEs */
    MPI_Bcast(&shmId, sizeof(shmId), MPI_BYTE, 0, MPI_COMM_WORLD);

    /*******************************************************************
    * Attach the memory segment to an address.  The rank 0 process will
    * request any address, then broadcast it to all other processes so
    * that they can attach at the same address.
    *******************************************************************/
    if (rank == 0)
    {
        if((shmVAddr = shmat(shmId, (void *)0, 0)) == (void *)(-1))
        {
            ExitError(rank, 2, "shmat error");
        }
    }

    MPI_Bcast(&shmVAddr, sizeof(shmVAddr), MPI_BYTE, 0, MPI_COMM_WORLD);

    /* Attach non-zero processes to address used by zero porcess */
    if (rank != 0)
    {
        if((shmVAddr=shmat(shmId, (void *)shmVAddr, 0)) == (void *)(-1))
        {
            ExitError(rank, 2, "shmat error");
        }
    }

    /*******************************************************************
    * Define an "arena" for shared memory location so that arena functions
    * (amalloc, afree, ..)may be used.  All processes must be linked to the
    * arena, but only process 0 must is required to keep an area pointer.
    *******************************************************************/
    if (rank == 0)
    {
        mktemp(shmFile);                /* Make file name shared arena */
```

```
    }

    /* Broadcast file name to everyone */
    MPI_Bcast(shmFile, strlen(shmFile) + 1, MPI_CHAR, 0, MPI_COMM_WORLD);

    usconfig(CONF_INITUSERS, maxUsers); /* Number of arena users */
    MPI_Barrier(MPI_COMM_WORLD);

    if ((usPtr = usinit(shmFile)) == NULL)
    {
        ExitError(rank, 3, "usinit error");
    }
    MPI_Barrier(MPI_COMM_WORLD);

    if (rank == 0)
    {
        ap = acreate(shmVAddr, shmSize, MEM_SHARED, usPtr, NULL);
        if (ap == NULL)
        {
            ExitError(rank, 5, "acreate error");
        }
    }

    MPI_Barrier(MPI_COMM_WORLD);
}
```

4.1.2 Allocated Space in Shared Memory

The GlobalMalloc functions is used to implement G_MALLOC. Once a shared memory arena has been set-up and made available to all processes, the GlobalMalloc function may use amalloc to allocate from that space, in the same fashion as malloc.

There are two intricacies in GlobalMalloc, the first is that only one process is allowed to call amalloc, otherwise, there will be multiple copies of memory allocated. After a single process (process 0) allocates, the memory, the memory address is broadcast to all processes.

The second intricacy is less important to the Ocean program, but may matter to others. A barrier is placed after the address broadcast to ensure that no process changes the content of the newly allocated memory until all processes have the address. For cases where the memory is used for things like signaling, it’s important that all processes be able to read it before it is first used.

The following C code implements the GlobalMalloc function:

```
void *GlobalMalloc(size_t size)
{
    int rank;
    static void *addr;

    /* Get process rank */
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    if (rank == 0)
    {
        addr = amalloc(size, ap);
    }

    /* Broadcast allocation to all processes */
    MPI_Bcast(&addr, sizeof(addr), MPI_BYTE, 0, MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);

    if (addr == NULL)
    {
        ExitError(5, rank, "amalloc error\n");
    }

    return(addr);
}
```

4.2 Lock Functions

The lock macros used by Ocean are: LOCDEC, LOCINIT, LOCK and UNLOCK. The implementation for LOCK and UNLOCK is discussed in the subsections which follow. The other two macros are straight forward to implement.

The lock functions that support PARMACS have the following functions:

- If multiple processes attempt to acquire a lock, only one may acquire it at any time.
- Processes waiting to acquire a lock must be granted the lock in the order which they requested it.
- Concurrent instances of different locks must be supported.

4.2.1 LOCK_TYPE

Variables of the type LOCK_TYPE are used to keep track of the status of each lock. A LOCK_TYPE variable contains two counters, nextAvailable and nowServing are used to implement a tick style lock. nextAvailable is the tick number which a process obtains when it waits to acquire a lock. The nowServing counter is the ticket number a that may use the lock. When a process' nextAvialable number matches the nowServing number, it may obtain the lock.

```
typedef struct
{
    int nextAvailable;
    int nowServing;
} LOCK_TYPE;
```

4.2.2 LOCK and UNLOCK

The procedures Lock and Unlock implement the PARMACS LOCK and UNLOCK macros. To acquire a lock a process enters Lock and atomically fetches and increments the locks nowServing counter. The value of the counter is store in a local variable. The process will then spin until the ticket value it obtained matches the locks nowServing number. When the two numbers match the process has acquired the lock. Several techniques like back-off and yielding the processor may be tried as potential optimizations to spinning.

To release a lock, a process calls the Unlock routine, and atomically increments that lock's nowServing counter.

```
void Lock(LOCK_TYPE *lock)
{
    int myTicket;

    /* Atomically increment next available and save result */
    myTicket = __add_and_fetch(&(lock->nextAvailable), 1);

    /* Loop until nowServing == myTicket */
    while(myTicket != lock->nowServing);
}

void Unlock(LOCK_TYPE *lock)
{
    /* Increment now serving and be done */
    __add_and_fetch(&(lock->nowServing), 1);
}
```

5 Results and Conclusion

The pthread translation for Ocean runs has been run on the GSL machines using Solaris, an Origin 2000 machine running Irix (modi4), and an earlier implementation executed on a DEC machine running what I believe to be Ultrix (I no longer have a means of running Ocean on the Ultrix machine).

Tests of Ocean on modi4 demonstrates nearly linear speed-up for fixed size problem and small number of (4 or less) processes. Figure3 depicts the average number of cache misses for the 258 × 258 Ocean, while figure 4 shows the average execution time in µs for the 258 × 258 Ocean. The X axis for both graphs are represent the number of processes (1, 2, 4, 8, 16, 32, and 64). The data for both graphs is the average of three executions. The Ocean code was modified to allow concurrency on an SGI Origin 2000. The function pthread_setconcurrency was used to request all processes run cocurrent, In every case (including the 64 process case) pthread_getconcurrency reported back the 1 level of concurrency per a processor (e.g. 16 for 16 processes).

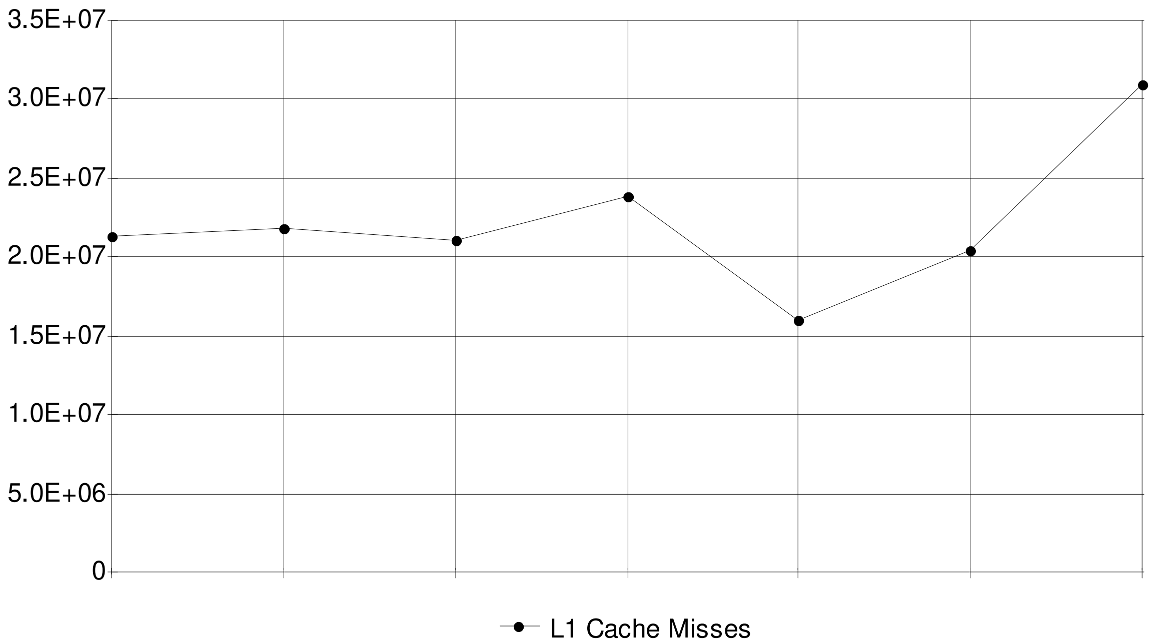

Figure 3 Level 1 Cache Misses

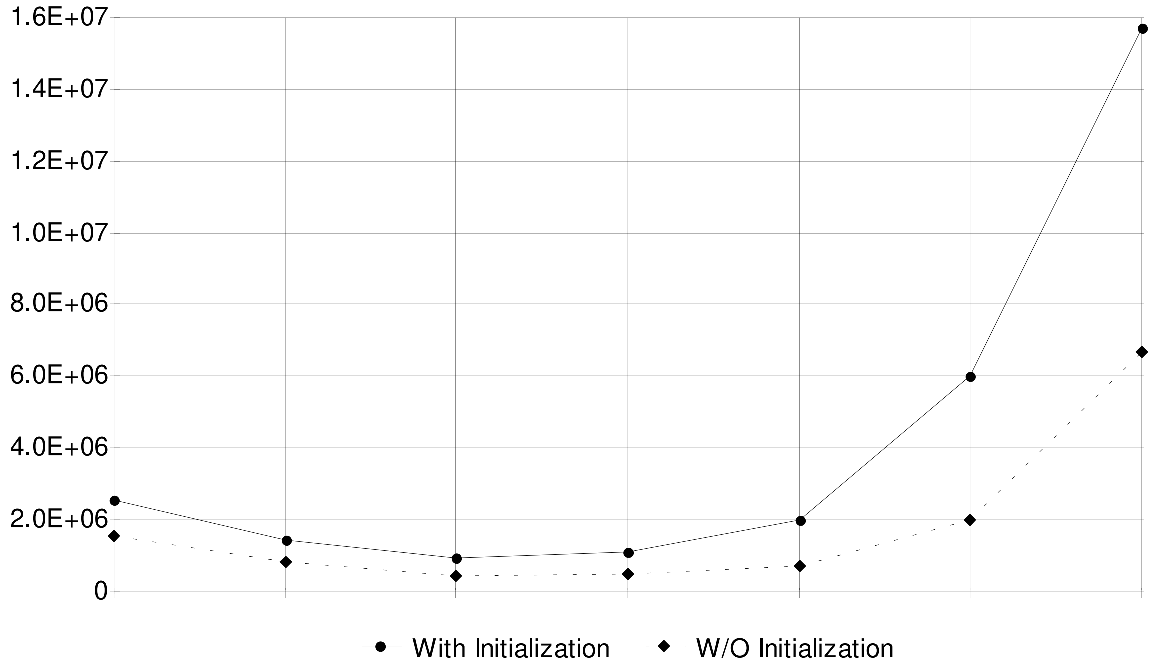

Figure 4 Execution time Including and Without Initialization

This lack of speed-up with 8 processes appears to correlate directly to cache misses, however, the cache miss level drops at 16 process, yet the execution time increases slightly. I believe the increased execution time is a result of increased communication between processes.

Stanford reports communication to computation ratio is sqrt(p)/N, so the amount of communication clearly increases with the number of processes. Increasing processes will also increase the time spent in barriers and is likely to increase contention over locked items.

I was unable to get Ocean to run correctly under MPI. I have a sample program (mtest) which demonstrates the correctness of the MPI code to support PARMACS macros. The problem of dealing with global variables has not been fully resolved. I have two versions of Ocean under MPI, both of which compile. The first one makes no attempt to handle the global variable problem and will run under any number of processes, but produce incorrect results. The second version attempts to handle the non-dynamic global variable problem, but aborts early on without any indication. It is likely that incorrect memory accesses are the cause of the problem.

6 References

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, Anoop Gupta, The SPLASH-2 Programs: Characterization and Methodological Considerations, International Symposium on Computer Architecture, June 1995

Silicon Graphics Incorporated, Message Passing Toolkit: Release Notes, Document 007-3689-003, 1999

Jaswinder Pal Singh, ANL Parallel Macro Processing Language, Fall 1996