[1] SplitThreads - Split-C Threads

Veena Avula, Induprakas Kodukula

Department of Computer Science
Cornell University, Ithaca, NY 14853

Abstract

SplitThreads are enhancements to Split-C by user level non pre-emptive threads. The primary motivation for *SplitThreads* comes from the fact that the SPMD paradigm of Split-C is a limitation for some important problems. At the same time, Split-C is very efficient and very tunable for good performance. Other related approaches such as *Nexus* have a large amount of overhead in providing threads capability. This paper presents the addition of a lightweight user-level threads package to Split-C. The performance numbers obtained show significant improvement over existing comparable approaches such as *Nexus*. The underlying thread library core is *QuickThreads*. This provides minimal support for thread management. Additional functionality is provided by *SplitThreads* on top of this core. Finally, *SplitThreads* provides higher level user objects such as I-structures and M-structures.

1. Introduction

Split-C provides SPMD model of programming. While this is sufficient for a wide variety of parallel applications, there is also a large class of applications which need threads. Some applications demand dynamic task allocation for reasonable performance; a few others demand multiple points of control. A threads package can provide us with precisely these facilities. From the point of view of functionality required by the user, a threads package needs to provide cheap synchronization, efficiently accessible global data structures and inexpensive scheduling.

This paper describes *SplitThreads* - an lightweight user-level threads package to Split-C. The performance numbers obtained show significant improvement over existing comparable approaches such as *Nexus*. The underlying thread library core is *QuickThreads*. This provides minimal support for thread management. Additional functionality is provided by *SplitThreads* on top of this core. Finally, *SplitThreads* provides higher level user objects such as I-structures and M-structures. The underlying communication library used is *Active Messages*.

The rest of the paper is organized as follows. Section 2 analyzes the software platform available for building *Split Threads* and the implications because of this. Section 3 describes the design of the thread management and scheduling routines. Section 4 describes the synchronization primitives and their design. Section 5 describes implementation of higher level user objects such as I-structures and M-structures. The performance of *SplitThreads* is contrasted with *Nexus* in Section 6. Section 7 describes the proposed applications. Section 8 describes future work and section 9 concludes the paper.

2. Software Platform

SplitThreads are implemented on the CM-5. The underlying software layer consists of a user level threads core

1. see http://www.cs.cornell.edu/Info/Courses/CS617/Split_C_threads/veena/index.html

package and a communication library. The threads core package is *QuickThreads*. It is a non-preemptive user level threads package with support for only thread creation and initialization. It provides no scheduling policies or mechanisms. It also lacks semaphores, monitors, non blocking I/O etc. It provides machine dependent code for creating, running and stopping threads. It also provides an easy interface to write and port thread packages. The higher level thread package (also known as the client) has the responsibility of providing any additional functionality. Since the *QuickThreads* package is very flexible, clients can be written which are easily tuned for specific applications.

Split-C on top of Active Messages provides a low communication overhead parallel programming language. It provides a global address space and an SPMD paradigm of programming. Its low-level nature and good modelling of the underlying hardware make it easier for a programmer to tune his application for good performance. Low cost synchronization primitives can be provided easily on top of Spilt-C because of the low overhead communication of active messages. This combination of low overhead of Split-C and the flexibility provided by *QuickThreads* makes this a very attractive platform to implement a higher level threads package with a good deal of functionality.

On a parallel machine, the threads package such as the one we have in mind should provide the functionality necessary for the application to view the entire machine as one giant processor. In particular, this involves providing the ability to create threads on specific nodes and on arbitrary nodes depending on the applications notion of data locality and load balance etc. This is because Split-C is primarily designed as a language to provide a great deal of flexibility to the user for good performance. The threads package should also provide global synchronization operations and user level scheduling primitives. Finally, it should also provide higher level global objects such as I-structures and M-structures. If the underlying software platform and the implementation are efficient enough then these higher level objects can be provided in an efficient manner.

3. Management and Scheduling

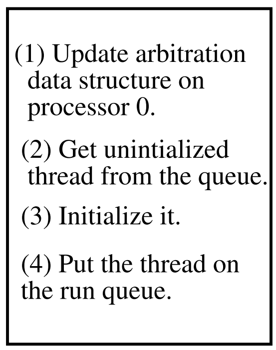
This section describes the thread management and scheduling routines.

3.1 Thread Management

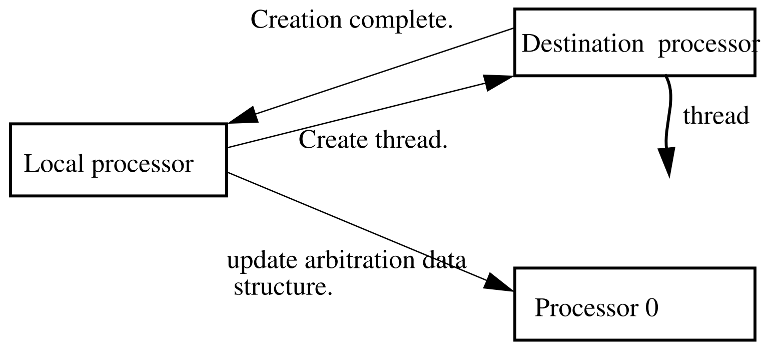
QuickThreads requires the client package to allocate a stack segment for every thread that needs to be created using the package. The size and the alignment of this segment are machine dependent. After the stack has been allocated, *QuickThreads* provides convenience routines that can be called to initialize this stack segment with arguments. *QuickThreads* provides support for arbitrary number of initial arguments using a varargs type of interface and a single argument for reasons of efficiency. After a stack segment has been initialized, the thread can be run anytime. QuickThreads library also provides support to threads to suspend themselves, allowing threads to save their state before suspension by running clean up functions. A thread starts executing with the function it was initialized to and runs till either it suspends itself or it aborts.

Active messages call a function handler upon receipt at a processor. To minimize the overhead of communication, it is a good idea to keep the handlers small. So, allocating memory in a handler is undesirable. As already mentioned, SplitThreads needs to have the ability to create threads remotely. A thread on a remote processor can be created by sending an active message with the appropriate handler and thread initialization arguments to the remote processor. This has two immediate consequences. First, the stack segment necessary for thread creation should not be allocated in the handler. Even if the stack were pre-allocated, it would still not be advisable to execute the thread because the handler would not return till the execution of the new thread completed. As a result, the execution of the handler would be unacceptably long in general and the scheduling of other threads would be interrupted.

So, in *SplitThreads* we decided to separate thread execution from thread creation. Every processor maintains two queues. One is a runnable queue of threads. These are initialized stack segments and are ready to run. The other queue is a queue of unintialized threads, but with allocated stack segments. On start-up, SplitThreads pre-allocates a pre-defined minimum of stack segments and adds them to the unintialized thread queue. On receipt of either a local or a remote request for thread creation, a stack segment is removed from the unintialized queue, initialized with the arguments passed by the thread creation routine and placed on the run queue. When threads complete execution, their stack segments are returned to the unintialized queue to be reused later. The number of uninitiated threads that can be created can be changed by the user at run time.

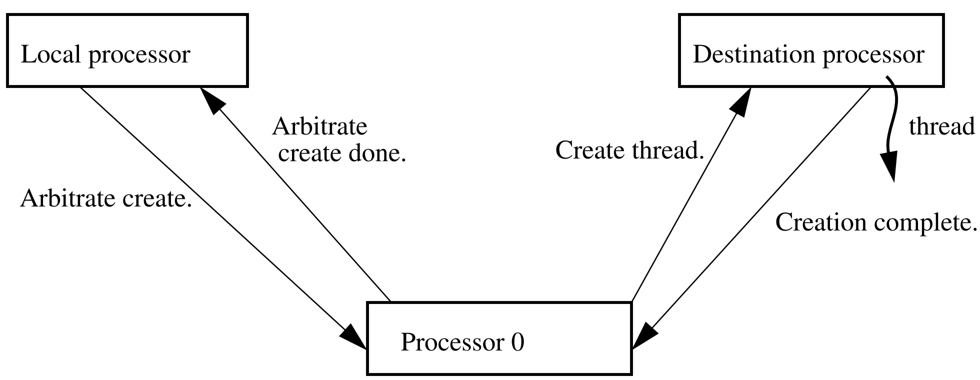


Local Create (Fig. 1)



Remote Create (Fig. 2)

The thread creation call comes in three flavors. It differs in on which processor is specified as the one for thread creation (also called the destination processor). On the CM-5, the processors are numbered from 0 to PROCS-1.



Arbitrate Create (Fig. 3)

MYPROC tells the local number of a processor. If the destination processor is the same as the calling processor, then the thread is created locally. If the destination processor is in the range 0..PROCS-1, but is not the same as the calling processor, then the thread is created remotely on the destination processor. This is achieved by sending an active message to the remote processor containing the thread initialization arguments. The handler of this active message uses the arguments in the message to initialize a thread from its set of pre-allocated threads and adds this thread to the run queue of that processor. Finally, the destination processor can be a wildcard processor (-1), in which case, the thread is run on a processor determined by load balancing criteria. In this case, a centralized arbitration algorithm is run on processor 0, which determines the processor with the lightest load and designates that as the destination processor for this thread. After this, the remaining sequence is identical to remote thread creation. It should be noted here that for this arbitration procedure to be meaningful, whenever threads are created on/by any processor, the arbitration data structures on processor 0 should be updated. This results in some additional overhead even in the case of local thread creation. It is not clear if a distributed arbitration procedure would have better performance than a centralized one. A distributed procedure would have to worry about consensus and might be quite expensive. Figure 1 summarizes the different types of thread creation.

3.2 Scheduling Routines

Creating a thread merely places it on a queue of runnable threads. The thread is not run. *SplitThreads* provides a convenience *start* routine which takes the first thread from the queue of runnable threads and does a context switch to it. Typically, this thread switches to other threads on the run queue directly after it has completed running. This is to minimize the number of context switches. The alternative would be to switch always to a "main" thread and then switch back to another thread on the run queue. By switching to a thread on the runnable queue directly, we save a context switch. This can be quite significant when the state saved for a thread is substantial.

However, the *start* routine as described above periodically requires the user to check if there are any threads in the runnable queue and if so, call it. This can be quite unwieldy. The solution we take in *SplitThreads* is to create an *idle thread* on every processor. The idle thread has the sole purpose of running forever and doing scheduling. If the idle thread ever runs, it checks if there are any threads that need to be initialized, and if so, places them on the run queue and puts itself at the end of the run queue. The idle thread is not counted for the purposes of load balancing.

It is also possible for a thread to voluntarily yield the processor to another thread. This is useful when the application programmer is aware that this particular thread is waiting for some event to happen. In such a case, the next runnable thread from the run queue is taken and run, and the yielding thread is put at the end of the run queue.

4. Synchronization

4.1 Mutexes

SplitThreads provides mutexes on a global machine wide basis. In other words, it is possible for a thread on one processor to acquire a mutex on a remote processor. This functionality is desirable because in a threads environment, different processors in a global address space environment can possess shared data of interest to other processors and it may be necessary to provide shared access to this data. This can be done by providing a mutex variable on the same processor as the shared data and making processors use this mutex to achieve access to this data.

Active messages can be used to implement mutexes in a very efficient manner. The basic idea is the following. Every mutex has an internal variable which maintains its state information. This state information describes whether the mutex is BUSY, i.e. owned by some thread currently, or FREE - i.e. available. It also maintains a circular queue of requests for it. This queue is a list of thread IDs waiting for access. A mutex is initialized to be FREE by the processor where it is located. To acquire a mutex, a thread calls a convenience routine mutex_lock. This routine sends an active message to the processor owning the mutex requesting access. If the mutex is FREE, then the sending thread acquires it. Otherwise, the requesting thread is enqued on the request list of the mutex. At the same time, the mutex informs the requesting thread to go to sleep. When the mutex is freed (by a call to mutex_release), the mutex wakes up the first thread (if any) on its request queue by sending an active message. The handler of this message wakes up the sleeping thread by removing it from the queue of sleeping threads and putting it in the run queue.

The nice feature of this implementation is that any particular lock operation costs at most 2 active messages - one to enqueue a request and another to get woken up. This is nice as it avoids a busy polling on the mutex. It is very easy to imagine a mutex becoming a hot spot (especially on a shared memory machine) as a result of contention between various processors to acquire it. A bad feature is that even though a thread is woken up, it is placed at the end of the run queue, and hence an acquired mutex may remain unused for a long time. This could be avoided by associating priority with threads and assigning higher priority to threads that were woken up as a result of acquiring some resource.

4.2 Join and Wait

SplitThreads provides routines for one thread to wait for the completion of another thread. These routines are similar in spirit to the *wait* and *join* mechanism provided for UNIX processes. A thread which creates another thread by a create call is called the parent of the second thread. The create routine carries as its argument the address of a memory location on the same processor as the creating parent thread where the created thread may store a value upon completion of execution. It achieves this by calling *join*. At the other end, the parent thread executes a *wait* routine to retrieve this value. The wait call blocks until the corresponding join call comes along. If the join call has already completed, wait returns at once. The memory location passed to the create routine can be used to communicate a return value and a return status.

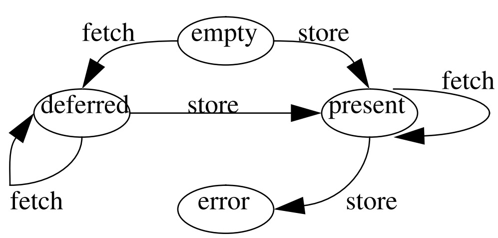
Again, active messages make possible an efficient implementation of this synchronization mechanism. Wait checks a memory location to determine if the corresponding join has occurred yet. If the join has not yet completed, then the calling thread suspends itself and transfers itself to the blocked queue. When the join occurs, it first checks to see if the parent thread already called wait and is on the blocked queue. If so, then in addition to returning the status and return value, the join message handler also “wakes up” the blocked parent thread by transferring it from the blocked queue to the runnable queue. Otherwise, it merely fills in the memory locations with the return value and status and exits. Wait saves processor resources by going to sleep rather than doing a busy wait. The same problem exists here that a woken up thread does not run at once, but this is relatively less important here as there are no precious resources that are being held unused.

5. I- and M- structures

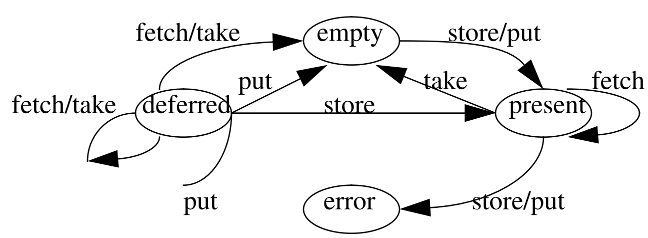
In this section, we describe higher level user structures such as I- and M- structures. One of the motivations is to examine how efficiently such higher level objects can be layered on top of the lower layer primitives provided by SplitThreads.

5.1 I- Structures

I-structures are write-once data structures and provide data structures with synchronization on a per element basis. Basically, every element in an I-structure starts out empty. Each element can be written atmost once. Reads of



I-Structures (Fig. 4)



M-Structures(Fig. 5)

an empty element are deferred until the element is written. If an element is already written then read returns the value present. Its an error to write to an element which is already written to.(fig 4).

In our implementation, I-structures are layered over mutexes. Every I-structure element has an internal variable which stores its status. The status can be either unintialized or initialized. If status is initialized then another internal variable stores the value the element is initialized with. Read on an I-structure which is not yet initialized results in the requesting thread getting added to a private queue of the I-structure element and the calling thread going to sleep. Read on an initialized I-structure element simply returns the value. When an I-structure element is initialized, its internal status is changed to represent this and all the threads in the queue waiting for the element's value are sent "wake up" active messages. This message transfers the thread from the blocked queue to the run queue in addition to returning the value of the element.

5.2 M-Structures

M-structures are mutable synchronizing data structures. Each element in an M-structure starts out empty. A write to an M-structure element updates its value if its empty. A read blocks if an M-structure is empty; otherwise it returns the value and empties the M-structure element. Only one read can complete after a write. It is an error to write to an M-structure element which is not empty. (fig 5)

The implementation of M-structures is quite similar to the implementation of I-structures. Read on an empty M-structure sends the calling thread to sleep and queues the thread on the M-structure's waiting list. A write to an M-structure element results in sending a "wake up" active message to the thread at the head of the waiting list (if any). At the same time, the value of the M-structure is reset to unintialized. A read on an M-structure with an initialized value returns immediately after "emptying" the M-structure element. Again, busy waits and hot spots are avoided.

6. Analysis of Performance.

In this section, we analyze send compare the performance of SplitThreads.

6.1 Thread creation

Local thread creation involves the four basic steps as listed in figure 1. It first updates the arbitration information on processor 0 (which is responsible for arbitration). Then it gets an uninitiated thread from the queue of unintialized threads, initializes it and places it on the run queue. The updating step is basically a global access to increment the count of number of threads residing on the local processor. Back of the envelope calculations show that a blocking update would take about 26us. Initializing the thread with it arguments is done by calling a *QuickThreads* routine and this takes 5us. Our measurements of the local create call with blocking update show that it takes 40us. However, we used non-blocking update for further optimization and this took 31us. The decrease in time here is because we are overlapping communication with computation. From our initial measurements of the local create we see that 9us is spent in the *SplitThreads* code. This is accounted by two accesses (the address and number of bytes) to check for remote data while initializing the arguments of the thread. Note that the thread creation (local, remote, arbitrate) calls take an unintialized thread and initialize its argument. So, they do not include time to allocate the stack itself. (This - stack allocation is done initially at start-up and later by the idle thread at run time. Also, the user has an option to allocate stacks for a specific number of threads at start-up).

The naive remote create would involve the following three main steps. It first sends an active message to the remote processor, the handler of which does a local create on the remote processor. Then, it would ship the arguments to the remote thread. Finally it sends a reply message to the local processor indicating the completion of the remote thread create call. *SplitThreads* has an interface to support varargs. This is done as follows. The arguments shipped to the remote processor are - the processor id of the local processor (which initiated the remote thread create) and a pointer to the data and size of data. The actual transfer of the data arguments should be done at the application level. (SplitThreads also provides a convenience routine which guarantees complete transfer of arguments at the time of thread creation. The performance of this would directly depend on the number of varargs the application program has for the created thread). However, the naive remote thread creation would spend 13us in the initial sending of active message, 40us (as calculated before) in the local thread creation and 26us + 26us to ship the arguments and 13us sending the reply message. This would total upto 118us. But, there are two major optimizations which can be done here. The first one is to use non-blocking access to remote data instead of blocking accesses. The second one is making the local processor take care of updation of the arbitration data structures while the remote processor is creating the thread (fig 2). So, the local processor would send message to the remote processor asking to do a local create, and would then proceed to update the arbitration structures on processor zero. The local thread creation on the remote processor would now only involve getting an unintialized thread, initializing it, putting it on run queue and sending a message to the local processor indicating the completion of remote thread create. This setup exploits the parallelism present in the remote thread creation procedure and also at the same time, overlaps communication with computation. It takes totally 58us which is very promising than 118us of the naive remote create.

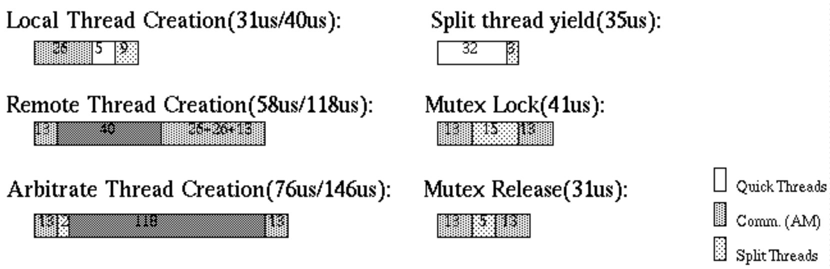
The measured times for arbitrate thread create are 76us. The arbitrate thread create basically involves sending an active message to processor 0, which then runs the arbitration algorithm and is responsible for remote creation of the load balancing thread, and sends a reply message to the initiating process indicating the completion of the creation of the load balanced thread (fig 3). This implies that this creation should take about 26us more than the remote thread creation and the results show that this is 76us which roughly matches the back of the envelope calculations.

Nexus provides similar functionality as *SplitThreads*. Comparison of performance for local and remote thread operations between Nexus and SplitThreads is given in the following table. Analysis of timings for local, remote and

Timing Measurements

Table 1:

	Split Threads	Nexus
Creation(local)	31us	150us
Creation(remote)	58us	100us
Context Switch	35us	120 - 250us
Mutex Lock	40us(rem)	30us(local)
Mutex Release	31us(rem)	30us(local)



arbitrate thread creations in the above figure is for the naive approach discussed earlier/

6.2 Mutex lock and release

The mutex data structure in *SplitThreads* consists of a status field (indicating whether the mutex is FREE or not) and a pointer to a circular queue of threads waiting to acquire the mutex. This queue is a circular queue of thread IDs. Note that the threadID in *SplitThreads* is unique for any thread and is sufficient to calculate the processor on which the thread is residing and identify it. Acquiring a mutex on a remote processor when it is FREE involves sending an active message to the remote processor, the handler of which would update the status of the mutex and send back a reply acknowledging the granting of access. This would mean a total of 26us is spent in communication. Measurements of the remote mutex lock show that it takes 40us. So about 14us is spent in the *SplitThreads* code. This is accounted by the fact that the handler to update the status of the mutex, would have to put in the new threadID on its circular queue. This is done by doing a malloc for the threadID and then inserting this in the circular queue. This dynamic memory allocation is responsible for a great part of the huge 14us in *SplitThreads* code. There are two ways of achieving better performance here. We can either do a pre-defined amount of allocation at start-up and later pass this responsibility to the idle thread or write our own dynamic memory allocation procedures.

Mutex release for a remote mutex would involve sending an active message to the remote processor, the handler of which updates the mutex data structure, send a message to both the next thread waiting on the queue (if any) and the local processor itself indicating the completion of the call. This analysis indicates that the communication involved would itself take 26us. Our measurements show that releasing a remote mutex takes 31us. This implies that around 5us is spent in the SplitThreads code.

6.3 Yields

Context switch in *SplitThreads* is done by calling *QuickThreads* routines. This context switch primitive first saves the register values of the old thread on to the stack of the old thread and adjusts the stack pointer. It then switches to the stack of the new thread, suspending the old thread. This (*QuickThreads* context switch) primitive should be called with the stack of the new thread and a helper function and arguments used to clean up the old thread once the context switch has been completed. Before the new thread is started, the context switch routine calls the client supplied helper function. Note that the helper function executes on the stack of the new thread.

The context switch routine of QuickThreads takes around 32us. Measurements indicate that the SplitThreads context switch takes 35us. This implies that 3us is spent in the SplitThreads code itself. This can be accounted by the additional state information introduced (per thread) by SplitThreads.

6.4 I and M structures.

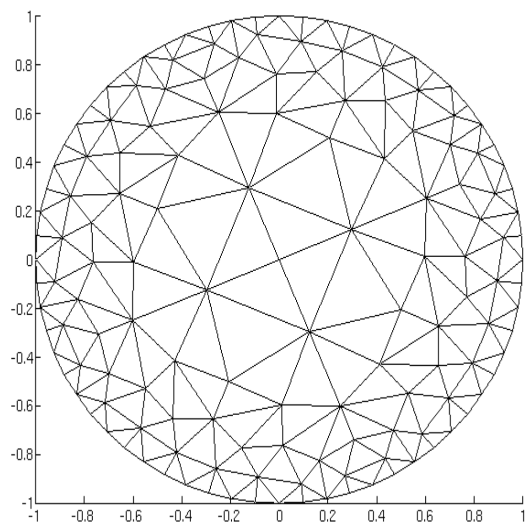
The I and M structures in *SplitThreads* are layered on top of *SplitThread* mutexes. Read of an I-structure element which is initialized involves a remote data access and would cost us a blocking read. On the other hand, read of an I-structure element that is not yet initialized depends on when the element would get initialized. Performance of the write of an I-structure element depends on the number of read requests queued for it. In the absence of any read requests queued up, an I-structure element write would cost us a blocking write. However, when there are reading requests waiting on an I-structure element, write would additionally involve sending "wake-up" messages to all the queued up threads. This in turn would depend on queued up threads are distributed i.e if all the queued up threads actually belong to one processor the wake up procedure would take longer than when these threads belonged to different processors.

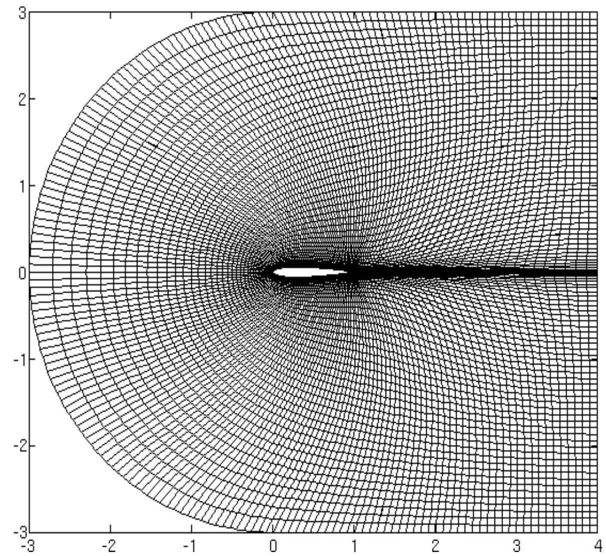
M-structure read when the element is initialized would be similar to the case of an initialized I-structured read. So is the M-structure write to an unintialized element which has no read requests queued up. Again, an M-structure read of an unintialized element would depend on when the element would get initialized. An M-structure write to an element which had queued up read requests would involve a blocking remote write plus a message sending the value to the first read request. This would be 26us + 13us. Performance measurements indicate that an M-structure write to an unintialized element which had at least one read request pending is about 40us. This means that the additional overhead introduced by SplitThreads in this case is only 1us.

7 Applications

7.1 Adaptive Mesh Refining.

In this section, we describe adaptive mesh refinement. This is a problem that is usually encountered in solving differential equations using a finite element method. A typical example is the flow of air around an aircraft foil. The differential equation corresponding to this is solved by formulating a finite element problem. The underlying mesh is derived by determining a mesh over the aircraft foil (shown on the right). It should be noted here that different parts of





this mesh have different density. In fact, very often it happens that the entire mesh is not known a priori. Instead an initial coarse mesh is generated by an initial partitioning. The differential equation is solved over this mesh and an estimate of the error is made. However, usually, for the accuracy of the solution desired, it is often the case that at particular places where the value of the function being solved for changes relatively rapidly even on smaller portions of the mesh. This is usually discovered by seeing that the error of the solution obtained is rather large in certain portions of the mesh. Then the application usually has to repartition the portion of the mesh over the area where the error is large. The point is that,usually a priori there is no way to find out which portion of the meshes going to be refined further. The refinement occurs as a result of dynamic results from the problem being solved. The load balancing is usually achieved by partitioning the mesh among the processors. Thus an initial load balance can be achieved by partitioning the mesh among all the processors. However, as adaptive mesh refinement occurs, it may easily happen that the processor loads get very unbalanced. So an SPMD programming paradigm could perform quite badly from a load balancing point of view on such a machine.

However, consider another approach. Every time a portion of the mesh is repartitioned, several threads are created which correspond to the units of work created by this new mesh partitioning. The threads are created in such a manner that each of them is created on the load on the entire machine. This makes load balancing much easier. Finally, in the presence of a low overhead threads package with user level scheduling primitives, it is expected that the price to pay for such load balancing would be quite small.

The figure at left indicates a circular mesh which has to be adoptively refined to evaluate the sum x^4+y^4. The center of the circle is at the origin. As a result, the mesh elements at the center of the circle are relatively large. However, as we move away from the center, small variations in the value of x and y result in much larger variations in x^4+y^4. This means that the corresponding portions of the mesh should be much finer. However, this can be determined only at run time, where portions of the mesh yield large values for the error and hence need to be recursively subdivided to yield better results.

7.2 Digital Circuit Simulation

Another application of interest to a threads package is digital circuit simulation. Logic circuits are simulated in software often before they are actually implemented in hardware. Most of current technology for these simulations is sequential and only currently work is being done to parallelize these applications. One logical way to simulate the circuit is to create a task for each circuit element. The sole purpose of this task is to sleep, wake up whenever it receives new input, and produce some output when it receives new input. It is natural to model these tasks as threads. There are several other reasons also. The entire processor is usually one address space. So, threads are more suited to this simulation than processes. Also, threads make the parallelization and load balancing more straightforward. Compare this with an SPMD approach. The number of circuit elements is typically in excess of 10^5. This makes it much harder to write an SPMD program and do management of the circuit elements in an efficient manner. Using threads simplifies this talk. Thread map onto circuit elements and these threads can be mapped onto processors for load balancing in a simple manner. However, in this case, the dynamic load variation is a lot less than in the case of AMR and the advantages of using threads are relatively lesser.

8. Future Work

One of the things that we would like to have is dynamic thread migration. With the present implementation, it is conceptually simple. All we have to do is transfer the stack from one processor to another. However, there are some issues such as the cost of such a transfer in terms of the underlying active message handler. But given the low overhead of the calls that we have provided, this may not be too bad. Also, even though we have pointed out applications that we believe will benefit from our implementation, we actually need to implement these. In particular, the type of adaptive mesh refinement and corresponding solutions to FEM methods seem to be of interest to a fairly large number of people.

9. Conclusions

Split-C provides an SPMD style of programming. This is enough for a wide variety of applications. It represents one approach to doing parallel programming. There is another perspective from another end to parallel programming. This is the functional programming/multithreaded approach such as ID-90. In such languages, the algorithm is specified at a very high level of abstraction, parallelism is unlimited and the main motivation is that by having potentially unbounded parallelism, it is possible to hide latency. However, these languages suffer from an excess of parallelism. In particular, it is not possible to exploit the enormous amount of parallelism in these languages on conventional machines easily. Automatic code generation from these languages by a compiler is hard.

The approach of Split-C is to provide the programmer an interface that is reasonably portable across various machines. At the same time, the programmer is exposed to enough details of the machine that the user can do a lot of fine grain optimizations to tune program performance. This is more easily done by the user than the compiler. For instance, the user can do things like scheduling communication keeping in mind when the data fetched will be really used. Split-C provides the user enough control over the execution of the program to make it easier for program optimization. In a sentence, the language can be described as one of small number of surprises.

SplitThreads are the implementation of a user level threads package on top of Split-C. It provides a low overhead integration of threads into Split-C. It supports load balanced dynamic thread creation with an overhead that is less than that of current methods. It separates the notion of thread creation from thread execution. It provides low overhead thread management routines. There are also thread synchronization primitives. Finally, it provides high level user objects such as I-structures and M-structures. From the implementation, and from comparison with other approaches (such as Nexus), if there is a lesson to be learnt, it is that by making correct design decisions, it is possible to achieve a great deal of functionality given a powerful and flexible underlying base. SplitThreads fully utilizes the active message technology to avoid busy waits (and thus improves processor utilization) and the thread library to provide fast context switches.

10. References

[1] David Keppel, Tools and Techniques for Building Fast Portable Threads Packages, *UW-CSE-93-05-06, Technical Report, University of Washington*, 1993

[2] David E. Culler, Andrea Dusseau, Seth Copen Goldstein, Arvind Krishnamurthy, Stevn Lumetta, Thorsten von Eicken and Katherine Yellick, Parallel Programming in Split-C, *Supercomputing*, 1993

[3] Ian Foster, Carl Kesselman, Steven Tuecke, Nexus: Runtime Support for Task-Parallel Programming Languages, http://www.mcs.anl.gov

[4] Veena Avula, SplitThreads Manual Pages, http://www.cs.cornell.edu/Info/People/veena/index.html