How I Spent Summer of 1993 at MCRC

Boon S. Ang

October 8, 1993

Contents

1 Introduction

This document describes the work I undertook as an intern at Motorola Cambridge Research Center in the summer of 1993. The work was supervised by Jamey Hicks.

The central theme of the work was to assess the effort needed to develop programs in Id. Attention was paid to the ease/difficulty of expressing the desired algorithms, and the efficiency of the resulting code, with emphasis on parallelism. In addition, the amount and type of tuning needed to get good performance from the code was also noted. The applications that were implemented included a parallel merge sort algorithm, and parts of a piece of speech recognition code (*mfcc_rec*) obtained from Victor Zue's Spoken Language Systems group at LCS. These will be described in detail below, together with performance results. I also re-implemented the Id I/O library several times to obtain better performance as I/O is used heavily in the speech recognition code. Lastly, I spent some time on an Ocean Circulation Model application which was, unfortunately, not completed by the end of summer.

Overall, we found that it is possible to obtain good performance with Id programs provided the algorithm has sufficient amount of parallelism. We found a number of issues that must be taken into consideration when developing algorithms and code in the Id system. These will be described in Section 2.

In terms of the ease of writing programs, we found that M-structures were needed on many occasions to achieve efficient implementations of certain functionalities such as hash tables and binary search trees. Undisciplined use of M-structures can make it very difficult to reason about the correctness of a program. The adoption of a number of guiding principles made this a manageable process. These will be elaborated in Section 3.1.2. M-structures are not always winners. We will show one such example in Section 3.1.3.

Tuning was required mostly to constraint memory usage so that the programs would run within the available memory on Monsoon. This is a known problem, and several researchers are working to solve it.

This work also suggested several areas where additional compiler optimizations can improve the generated code, particularly with respect to higher order function applications. These are mentioned in Section 4.

2 Parallel Merge Sort

This work was motivated by the desire to write a sorting routine in Id which has good parallelism, and short overall run time. The latter means that the algorithm and implementation must not involve large constants in their time complexities. The development of this algorithm illustrates the issues that one should consider when writing Id code.

Central to our timing considerations are the time complexities of: (1) *total work*, (2) *critical path length*. The ratio of the former to the latter time complexities yield the order of parallelism. Space complexity is another important consideration. Unfortunately, this work was unable to address the issue in any depth due to lack of time. The only place where this was taken into account in the Parallel Merge Sort code was to restrict the breadth-first expansion of the recursive call tree so that beyond a certain depth, a depth-first recurisve-call approach is taken. This was implemented in user code and was necessary to avoid running out of frame storage.

To stay competitive with sequential code, we need an algorithm with *total work* time complexity which is the same, or very closed to that of the best sequential algorithm. In addition, the algorithm should have good parallelism. To this end, it may be worthwhile to trade off a small increase of say $O(\log n)$ amount of work if that can increase parallelism by a large amount. Thus, for the problem of sorting n floating point numbers, we would prefer an algorithm that has total work of $O(n \log n)$, but would accept an $O(n(\log n)^2)$ algorithm if that is necessary for achieving decent parallelism.

Determining the complexity of the critical path length requires consideration of the longest data-dependence path. This is not always easy but for a well defined problem with highly structured solutions like sorting, this is not too difficult. As examples, we will consider the critical path length of several classical, mostly sequential, sorting algorithms.

First, let us look at *quicksort*. In its generic form, quicksort picks a pivot element out of the set of elements to be sorted, and partitions the remaining elements around the pivot element. It then makes two recursive calls, one with the set of elements smaller than the pivot element, and the other with the set of elements greater than or equal to the pivot element. If done in the straight forward manner, partitioning is sequential, with a data-dependence path on the order of the length of the original set of numbers. Thus, the very first partitioning done in this manner will introduce an $O(n)$ data-dependence path. In addition, in a simple implementation that uses arrays to keep track of the various sets of elements, Id's ability to pipeline producer/consumer computation through the use of I/M structures does not help as the partitioning at the next level needs to know the size of the set of element that it is partitioning, and this is not known till the previous partitioning is finished. A list implementation allows more pipelining, but requires manipulation of `cdr` pointers, an added overhead. Regardless of the data-structure adopted, a naive encoding of quicksort in Id has an $O(n)$ critical path, and at most $\Omega(\log n)$ parallelism, assuming the average case total work of $\Theta(n \log n)$.

Next, let us consider mergesort. While Id's pipelining of computation through I structures is helpful here, allowing overlap of merging at different depths of the call tree to happen concurrently, the most straight forward way of merging makes each merge sequential, and will again introduce a data-dependence path on the order of the size of the resulting set. Thus, this is again not good. Parallelism is still at most $\Omega(\log n)$.

The above shows that while it is possible to write these common sorting algorithms in Id, consideration of the order of their critical path lengths shows that they will not yield good performance as the amount of parallelism is at most $\Omega(\log n)$. Analysis similar to the above is necessary for an understanding of the potential parallelism an algorithm. Simply coding up the algorithm and looking at a few performance numbers is not a substitute although they are important for confirming the analysis results.

We next considered several well known parallel sorting algorithms but most of them are not attractive for implementing in Id. Bitonic sort, for example, while having good critical path length of $O(\log n)^2$ has total work of $O(n(\log n)^2)$. It is also quite complicated, and the constants hidden in the complexity terms are not exactly small. The other parallel sorting algorithms that we look at, such as Odd-Even Merge sort, are mostly meant for the SIMD computation model. They are therefore tuned for a different set of constraints, and are often not straight forward to code up in Id.

We finally decided to modify merge sort to make the merging parallel. Once this is done, the algorithm becomes fairly parallel, while retaining a small constant in its time complexity. The algorithm is simple and straight forward to code up in Id. While we developed the algorithm on our own, we found out later that a highly similar algorithm was described in [2].

2.1 Algorithm

The overall sorting algorithm is based on merge-sort, with the difference that a parallel merging algorithm is used. We will first describe how parallel merging is done, and then revisit the overall algorithm while analyzing the time complexities.

2.1.1 Parallel Merge

Suppose we want to merge 2 arrays A and B of about the same size, and we would like to obtain P-fold parallelism. The parallel merging algorithm first divides A into P contiguous pieces. Let the first (i.e. smallest) element in each piece be a_0, a_1, $\ldots$, a_{p-1}.

$a_0 \ldots$	$a_1 \ldots$	$\ldots$	$a_{p-1} \ldots$	A

Next, we find for each $a_i, 0 < i \leq (p-1)$, the corresponding element in B, let's call it b_i, such that either ($b_i = b_0$ and $a_i \leq b_i$) or ([b_i's left neighbor] $< a_i \leq b_i$). If no such b_i exists, use the "phantom" B element, b_{n+1} as the b_i. For a_0, the corresponding B element is the first element in B, b_0.

$b_0 \ldots$	$b_1 \ldots$	$\ldots$	$b_{p-1} \ldots b_n$	B

Assuming a model that allows concurrent reads (we will address this assumption again later), each b_i can be found in $O(\log n)$ steps using a binary search, and we can perform this search for all P b_i's in parallel. Once this has been done, we have divided A into P subarrays, and similarly for B. We can then merge the P pairs of A and B subarrays in parallel, using a sequential merging algorithm for merging each pair.

The one catch here that prevents us from getting P times speedup is that the split into P pairs of subarrays may result in B subarrays that have very uneven sizes. If there are B subarrays that are too big, say there is one that is close to the size of the entire B, the overall merging process will have a long critical path. We get around this by checking the sizes of the B subarrays. For each B subarray that is too big, for example greater than a small constant times the perfectly even size, we split it up evenly into enough pieces to make each new chunk small enough. We then find the corresponding A elements just like above (interchanging A and B).

To summarize, we first divide A into P even chunks, and find the corresponding B chunk. If any B chunk turns out to be too large, we divide that into small enough pieces, and find the corresponding subdivision of the A chunk. We may thus end up with more than P sequential merges to perform, but that is not too much of a problem for Id as the overlap between different depts of the recursive call tree made possible by I-structures should absorb such slight unevenness. In general, we can be sure to obtain $O(p)$ parallelism from this merging algorithm. The critical path is $O(n/p + \log n)$. P can therefore be as large as $\Theta(n/(\log n))$ while being work efficient (Tom Leighton's notion of work efficiency).

2.1.2 Parallel Merge Sort

A space efficient way of implementing the parallel merge sort using the parallel merge described in the previous section is to unfold the binary recursive call tree until we have approximately P leaves. We then sort the numbers in each P leave with the most efficient sequential sorting algorithm. Finally, we perform parallel merge up the call tree starting from the P leaves.

Time Complexities

We will first analyze the time complexities assuming a model with constant network latency, and a heap that allows concurrent access. These constraints will be relaxed subsequently.

First, let us consider the sequential sorting part, and find its the critical path length. The sequential sort is performed on $\frac{n}{p}$ elements. This takes $O(\frac{n}{p}\log\frac{n}{p})$ time. The parallel part of the call tree is $O(\log p)$ deep. At each level of the call tree, we perform parallel merges, each of which is made up of a split and then several sequential merges. The split takes $O(\log n)$ time, while the subsequent sequential merges, $O(n/p)$ time.

The overall critical path length is therefore $O(\frac{n}{p}log\frac{n}{p}) + O((\log p) * (\log n + \frac{n}{p})))$. Total work remains, as in the sequential case, at $O(n \log n)$, which is nice. We can thus efficiently utilize P up to $\Theta(\frac{n}{(\log n)})$. With P at $\Theta(\frac{n}{(\log n)})$, critical path becomes $\Theta(\log n)^2$.

A more reasonable network latency is $O(\log P)$ instead of constant. This holds for butterfly type interconnection network. The effect of this on the above time complexities are that we can efficiently make use of up to only $P = \Theta(\frac{n}{(logn)^2})$ processors, with critical path of $O(\log n)^3$. Total work remains unchanged.

One last concern is our assumption that the heap memory supports concurrent accesses. This is not a realistic assumption. In particular, the split during a parallel merge makes use of concurrent accesses in constant time to achieve the above time complexities. In this work, we made an attempt at performing the split in an exclusive read model while preserving the same timing complexity. By using a *double binary split* algorithm which we will describe in the next paragraph, we can guarantee that we do not have more than $O(\log n)^2$ accesses to the same location. This has the effect of making the critical path a factor of $O(\log n)$ longer compared to the concurrent read, constant network latency model. If we are already including a factor of $O(\log n)$ network latency however, this factor added for network latency can also account for the effects of multiple accesses to the same location in an exclusive read model. Thus, a model with exclusive reads, and $O(\log n)$ network latency can efficiently make use of up to $P = \Theta(\frac{n}{(logn)^2})$ processors, with a critical path of $O(\log n)^3$.

Double Binary Split

We will now describe the double binary split algorithm. Supposed we have P ordered numbers a_0 to a_{p-1}, and an array B with elements b_l to b_u. We want to find the b_i's that "match" the a_i's. Instead of finding the match for each a_i independently, we start by comparing a_m, where $m = \frac{1}{2}p$, with the middle element of array B, b_{mid}. If $a_m \leq b_{mid}$, we make two recursive calls: one with the subarray of A, a_0 to a_m, and B from b_l to b_{mid-1}; the other with A from a_{m+1} to a_{p-1} and B from b_l to b_u. If it turns out, however, that $a_m > b_{mid}$, we still make two recursive calls, but with A from a_0 to a_{m-1} and B from b_l to b_u; and then with A from a_m to a_{p-1} and B from b_{mid+1} to b_u. One additional change needs to be done to achieve the above mentioned upper bound of $O(logn)^2$ on the number of accesses to the same location. We need to pass the middle element of B, b_{mid}, from caller to callee to avoid having to re-read that every time.

2.2 Run Time Statistics

We coded up the above algorithm in Id and ran it on the Monsoon hardware at MIT. We achieved "depth-first" computation after the recursive call tree has produced P leaves by using a barrier to separate the two recursive calls in Parallel Merge Sort. We tried both merge-sort and quicksort as the sequential sorting algorithm. Quicksort was expected to give better performance, but only so with an un-released version of the Id compiler. Even then, the improvement was not great. The statistics below were collected using merge-sort as the sequential sorting algorithm. We used I-structures and no M-structures. To reduce the number of calls to the heap manager, we allocated $(\log n)$ I-structure size n arrays, one for each depth of the recursive call tree, into a $(\log n)$ element I-array. The calls at each depth can then fetch the array for that depth, and work on non-overlapping portions of the array.

To show that the algorithm scales with problem size, we sorted 2,000, 10,000 and 100,000 floating point numbers generated by a random number generator. The statistics for the 8pe8is configuration are shown in Table 1. Even with only 2,00 numbers, the algorithm showed good parallelism, with only about 25% idle cycles. The algorithm performed better as the problem size goes up. The same could not be said of algorithms like the generic merge-sort where the critical path length is $O(n)$. Indeed, execution profiles that we obtained on a 4pe configuration Monsoon showed a tail with low parallelism at the end of execution that got worse as problem size increased.(Diagram 1)

We also collected some statistics to show scaling with machine size. We did not go all the way from 1pe to 2pe, then 4pe and 8pe, mainly because we ran into some problems running the code on 2pe: it locked up the network. Instead, we only collected statsitics for 4pe and 8pe configurations, which are displayed in Table 2. The statistics shows that we achieved practically linear speedup in going from 4pe to 8pes. The P that we used for these runs are $8\times$(number of PEs) because of Monsoon's 8 way interleaved pipeline.

3 Speech Recognition

We obtained the *mfcc_rec* speech recognition code, written in C, from Victor Zue's research group at LCS and recoded the Viterbi Scoring part in Id. Mfcc_rec is the speech recognition component used in speech understanding systems developed by Zue's group, e.g. Voyager.

The original code takes in digitized speech input and a language model, and outputs the

size	Statistics	
2,000	Utilization	2,178 $\times 10^3$cycles
	Idle	25.57%
	Total	2,927 $\times 10^3$cycles
	Critical Path	366 $\times 10^3$cycles
10,000	Utilization	11,570 $\times 10^3$cycles
	Idle	18.13%
	Total	14,141 $\times 10^3$cycles
	Critical Path	1,768 $\times 10^3$cycles
100,000	Utilization	132,014 $\times 10^3$cycles
	Idle	16.17%
	Total	157,470 $\times 10^3$cycles
	Critical Path	19,684 $\times 10^3$cycles

Table 1: Scaling of the algorithm with problem size.

Figure 1: Profile of generic merge-sort, sorting 100,000 numbers on 4pe4is.

Statistics	4pe4is	8pe8is
Utilization	131.5 $\times 10^6$cycles	132 $\times 10^6$cycles
Idle	16.57%	16.17%
Total	157.7 $\times 10^6$cycles	157 $\times 10^6$cycles
Critical Path	39.4 $\times 10^6$cycles	19.7 $\times 10^6$cycles

Table 2: Sorting 100,000 numbers using the Parallel Merge Sort algorithm coded up in Id and ran on Monsoon.

n sentences among the permitted utterances (as defined by the language model) which the program considers as the best matches for the input speech. Mike Phillips, a researcher in Zue's group suggested that the Viterbi Scoring part is a good candidate because it is time consuming, should have good parallelism, and the time taken grows with the complexity of the language model (e.g. with vocabulary size).

For this project, we ran the C code and dump out the required data just before the Viterbi Scoring part. The output data was then fed into the Id Viterbi Scoring code. By comparing the scores, we were able to confirm that the two pieces of code performed the same function. The Viterbi Scoring code is followed by an A* search before the n sentences are determined. We were originally hoping to code up the A* search too, but we did not manage to finish that by the end of summer. The report will therefore be dealing mostly with the Viterbi Scoring code.

3.1 Viterbi Scoring Code in MFCC

The Viterbi Scoring code performs a similar scoring function as that done in conventional Hidden Markov Models (HMM). We will not explain the details of Hidden Markov Models here. Readers who are not familiar with HMMs are referred to [1] for an excellent tutorial which assumes no prior background.

The Viterbi Scoring performed in mfcc_rec differs from classical HMM's in several ways. One is that it uses a segment model, where transitions occur in segments that can be of variable time duration in the input utterance. The model also allows insertions and deletions. However, all these differences are rather minor departures from the classical HMM code.

The language model defines a set of inter-connected nodes that represent *phones*. The Viterbi scoring occurs across the duration of the utterance. For each time step during the utterance, it computes the score for every node in the model. A typical model for a vocabulary of 500 words has about 17,000 nodes. As the update within each time-step can be done in parallel, there is potentially a lot of parallelism during Viterbi Scoring. Computation of the scores at a particular time step requires the final scores of nodes at earlier time steps. There is therefore data-dependence between time steps that sequentializes computation. However, the performance reported in the next section shows that there is no problem keeping the largest configuration of Monsoon, which requires 64-fold parallelism, busy.

Category	Statistics ($\times 10^6$ cycles)			
	1pe1is	2pe2is	4pe4is	8pe8is
Utilization	4,411	4,468	4,498	4,511
Idles	705	1,086	1,510	1,122
Total	5,116	5,554	6,008	5,633
Critical Path	5,116	2,776	1,502	704
Percentage non-idle	86.2%	80.4%	74.9%	80.1%
Speedup	1	1.8	3.4	7.27

Table 3: Run time statistics of the Id Viterbi code running utterance input 2g0011ss.wav on different machine configurations.

3.1.1 Performance

We report the performance statistics for a version of the Id Viterbi code that uses both I- and M-structures for keeping scores. A purely I-structure version is preferred but requires substantial departure from the original C code. A fully M-structure version that is a direct translation of the C code was implemented earlier, but found to have somewhat worse performance than the hybrid I/M-structure version.

Table 3 shows the execution times across different machine configurations. The statistics were obtained in the following manner. Firstly, we removed the time required for reading in the data. This was done because the input part is highly sequential and shows little speedup when we move to larger machine configurations. As not much attention has been paid to making Id I/O parallel, we feel that we should exclude I/O from the current study.

Owing to the variability of I/O times, we went to some length to ameliorate the inaccuracy introduced by it. We first ran only the input code to obtain the run-time for the input portion of the program. As this varied over runs, we took the average of several runs. In addition, we performed the same Viterbi Scoring several times after reading the input once whenever possible. (A barrier separates the input code from the rest of the program.) This reduces the impact of inaccurate I/O timings by dividing the variation in input timing by the number of times Viterbi Scoring is performed. Unfortunately, with small machine configurations, we were not able to perform Viterbi Scoring that many times before running out of heap memory. On 1pe, we only performed it once. For 2pe, 4pe, and 8pe, we ran the Viterbi Scoring 3, 7 and 10 times respectively. The statistics for the larger machine configurations were the ones that were most sensitive to the variations in input time. This was because the relative input time to Viterbi Scoring code time decreased with larger machine configurations as the latter decreased while the formal stayed very much the same. Thus the fact that we could only make a few Viterbi Scoring runs for the smaller machine configurations does not matter that much.

Overall, we find that speedup is good for this version of the program.

3.1.2 Using M-structures

The use of m-structures is motivated by the desire to avoid consing up heap memory as frequently as similar code implemented using only functional or I-structures. In certain applications, they also avoid the "threading" that occurs with the use of functional structures. M-structures, however, have to be used in a disciplined way to keep them manageable. The use of m-structures usually requires some form of synchronization in the user's code in order to ensure meaningful execution of the program. This may involve coordination of different parts of a program. The undisciplined/unstructured use of m-structures will make this very complicated, and the resulting code hard to read and change.

An important tool in making the use of m-structures manageable is to hide an m-structure implementation underneath an interface that guarantees some *execution-order-invariant properties*. Any such undertaking, however, requires both the design of a good set of interface functions and the efficient implementation with m-structures. The latter usually means adopting synchronization strategies that do not have large critical sections, thus ensuring a sufficient amount of parallelism.

We will cite an example of a use of m-structures here. In the Id mfcc_rec code, we use a hash-table which is implemented with an m-array of list of entries. The m-array implements the buckets of a hash-table, with conflicts resolved with the list. We could have used m-lists instead of lists, and lock coupling down a list to allow concurrent pipelined access to each list. However, as we expected to keep each list small, and performing lock coupling with m-lists is more expensive than simply going down a list, we opted for simple lists.

A key design feature is the interface which is based on the invariant that a lookup in a hash-table will return the value with the given key as long as such a value is eventually entered into the hash-table. The hash-table implementation allows the lookup to arrive earlier in time during an execution then the storing of the value with a matching key and yet obtain that value. This requires that values can only be added to a hash-table but not removed. (This is the "monotonically increasing" property that Burton Smith talked about.) Also, only one value with a particular key can reside in a hash-table.

The design also includes a feature for closing the hash-table. This allows one to declare that no more values will be entered into the hash-table after a point in execution. With this feature, one can check for lookups that have not been satisfied at closing, which will not be satisfied by further additions to the hash-table. The interface can then allow one of several things to happen. One is to declare an error if the programmer does not expect such a situation to arise. Alternatively, the user can supply a default value to be returned. Some sort of value has to be returned for each outstanding lookup as the lookup code was expecting a reply.

To sum up, the key is to hide M-structures behind an interface that can cope with non-deterministic order of execution while providing some invariants that allows its functionalities to be well defined and useful. Once we have the interface, the underlying implementation can be pursued independently, and indeed changed over time, perhaps starting with a simple synchronization protocol that provides small amount of parallelism and gradually improving the parallelism with more sophisticated implementations. Most of the ideas mentioned here are not new, but it is interesting to see how they can make the use of m-structures much more manageable.[1]

3.1.3 M- versus I-structures

The use of M-structures can enable finer-grain concurrent execution, but can also hinder parallelism by requiring synchronization at a coarser granularity. This can be particularly problematic if the synchronization occurs at the top level of the program.

We observed such a situation in the Id Viterbi code. At the top level, we have a loop that iterates over the time steps of the utterance. During each time step, the scores at each node is updated many times through several different processes. While some of these updates can occur in parallel with take-put synchronization, some of them have to occur sequentially, one after the other has completed. With the M-structures implementation, the notion of completion has to be applied across the entire array of scores. Thus synchronization occurs at a very coarse granularity. This brings with it the problem that as we approach the synchronization point, parallelism decreases. It also takes some time for parallelism to ramp up again after that.

We observed this problem with the purely m-structure implementation of the Viterbi code. We saw periodic blinking of the panel lights on Monsoon at the synchronization points. It is also reflected in the statistics where the number of idle cycles is significant. When we reorganized the Viterbi code so that some parts use I-structures, we observed less blinking of the lights, and fewer idles cycles. In this particular application, I-structures allow more consumer and producer overlap of execution than M-structures. The moral of the story is that if the code can be expressed fairly efficiently with I-structures rather than M-structures, it is better to use I-structures. It is also worth mentioning that the use of M-structures necessitated the inclusion of barrier synchronization at critical places. This leads to code that are harder to get right in the first place, and difficult to alter subsequently.

3.2 Id I/O

The Id I/O library (trig_io) was re-written several times to improved its efficiency. This had to be done before other code development because we needed to read in input data which ran into several megabytes. When we started using the I/O library, we ran out of memory while trying to read in the needed data, even on a 4pe4is machine which has 16 mega (64 bit) words of memory. This occurred after about 40 mins of execution. An investigation into the problem revealed that this was due to tuples (heap objects) that were consed up during I/O calls which were then discraded but not recovered.

We got around this problem by doing two things. We enhanced the MRV optimization in the Id compiler to handle more cases, removing the need to cons up tuples in many instances in the I/O code. The enhancement has to do with allowing the MRV conversion to occur across seq encapsulators where it is safe to do so. (This description is targeted towards readers who know the Id compiler well. You can safely ignore it if it does not make any sense to you.) The other change involved the restructuring of a large part of the trig_io I/O

[1] I would like to thank Shail Aditya for sharing many of his experiences in using m-structures in the Id in Id compiler project.

library to make the code cleaner, require fewer procedure calls, and also make the code more amenable to MRV optimization.

With all these changes in place, we were able to read in all the data in about 15 mins on a 1pe1is system which has 4 mega-words of data. We next added support for binary I/O, and converted the input data into binary format. This removed the need to parse the input into integers, which is both highly sequential and work intensive. We also added double buffering, which allows data to be read out of one buffer by user code, while a second buffer is filled by the front-end. Finally, we changed the library code to make use of I/O channels on all the front-ends of any given machine configuration. The previous I/O library only used I/O channels on PE 0.

With all the changes in place, we were able to read in all the data in about 2 minutes on a 1pe1is Monsoon. Most of the time is idle cycles (over 90%), suggesting that most of the time is spent waiting for the front-end to fill the input buffer. With a 4pe4is Monsoon, which has two front ends, if the two large input files are opened on different front ends, we observed a speedup of about 2 (just a little over a minute to read in the data).

It is interesting to note that in order to achieve true double buffering, we had to implement a background job server. This allows the refilling of a buffer to occur in the background without hindering the progress of other user code. The same solution, being implemented by Jamey in Monasm, will be used to solve a long outstanding problem with Global Constants and Thunks in the Id/Monsoon system.

3.3 Incomplete Work

We started implementing the A* search but did not get very far. A preliminary implementation of the A* search "stack", which is actually a prioritized queue, has been completed. This implements the queue as a binary search tree. It uses M-structures and guarantees that each update to the "stack" is atomic. Supported updates to the queue includes enqueuing, extracting the best, or extracting the worst. Concurrent updates is achieved while preserving atomicity by using lock coupling down the path from the root of the tree to the node of interest.

The greater issue that has not been dealt with, with respect to the A* search is how to make the search parallel. The A* search basically extends the search along the path that is most promising at any moment. A* search also guarantees certain properties, like admissibility if the scoring functions meet certain conditions. All these have to be taken into account when we implement a parallel version of the A* search. There is a sizable body of literature concerning parallel search which should be helpful. However, we did not get around to investigating them.

4 Ocean Circulation Model

We will keep the description here short. The original code for the Ocean Circulation Model was developed by Professor Arvind and Professor Marshall. I made some additions and changes, and also studied the almost complete code extensively. There are several things worth reporting here.

The first concerns the structure of the the computation. The program models the ocean as a 3-D matrix of cubes, which maps well onto a three dimensional grid of processors. With such a mapping, communication is mostly between neighbors in the 3-D grid. This structure should be taken advantage of, even on a Fat Tree connected machine like *T, to cut down communication across the network (mapping neighboring grid points to the same processor when the number of ocean grid points is larger than the number of processors), and to keep communication within a small neighborhood when it has to occur over the network.

The second thing we saw in this code concerns the use of high order functions and the way they are currently compiled in Id code. Firstly, currying made even the *full* application of higher order functions expensive, partly because of the naive way these are currently compiled by the Id compiler. As many applications of higher order functions are actually full applications, we should first check if we have a full application before falling back to building closures. A better improvement will be to specialize code and propagate constants, including function arguments across procedure call interfaces to the specialized code. This will remove a lot of higher order applications. The latter optimization is also needed to achieve efficiency in modules/packages like sorting routines where the comparison function, and key access functions are passed into generic sort functions as arguments.

5 Conclusion

In conclusion, we implemented several pieces of code in Id this summer, gained some experience and insight into the use of M-structures, and also saw many places where the Id compiler can be changed to generate better code.

References

[1] L. R. Rabiner. A tutorial on hidden markov models and selected applications in speech recognition. In *Readings in Speech Recognition.* Morgan Kaufman, 1990.

[2] Y. Shiloach and U. Vishkin. Finding the maximum, merging, and sorting in a parallel computation model. *Journal of Algorithms*, 2:88–102, 1981.