References

[1] G. S. Almasi and A. Gottlieb, *Highly Parallel Computing*, Benjamin/Cummings Publishing Company, 1989.

[2] K. E. Batcher, "Sorting Networks and Their Application," *Proc. AFIPS 1968 SJCC*, pp. 307–314, 1968.

[3] G. E. Blelloch, C. E. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha, "A Comparison of Sorting Algorithms for the Connection Machine CM-2", *Proceedings 3rd Annual ACM Symposium on Parallel Algorithms and Architectures*, pp. 3–16, 1991.

[4] X. Guan and M. A. Langston, "Time-Space Optimal Parallel Merging and Sorting," *IEEE Transactions on Computers*, Vol. 40, pp. 596–602, 1991.

[5] B-C Huang and M. A. Langston, "Practical In-Place Merging," *Communications of the ACM* Vol. 31, pp. 348–352, 1988.

[6] R. M. Karp and V. Ramachandran, "A Survey of Parallel Algorithms for Shared-Memory Machines," Technical Report UCB/CSD 88/408, Computer Science Division, University of California at Berkeley, 1988.

[7] C. P. Kruskal, L. Rudolph and M. Snir, "A Complexity Theory of Efficient Parallel Algorithms," *Theoretical Computer Science*, Vol. 71, pp. 95–132, 1990.

As already noted, a significant drawback of the S81 is its synchronization overhead. The less tasks must cooperate with each other, the better suited the algorithm is for MIMD execution. The transferral of data between processors will necessitate some amount of synchronization. Barrier synchronization is, perhaps, the wrong type of synchronization primitive to be used in non-numeric parallel algorithms. A better synchronization scheme would be to have only those processors that are cooperating with each other synchronize together. Alas, programming, and debugging, with such synchronization could easily become a monumental task. Distributed-memory MIMD machines can employ their message passing facilities to synchronize only those tasks that need to cooperate closely as well as share data between processors. The connection network of such a system would also be able to move more efficiently large amounts of data between processors.

When algorithms call for very close cooperation between processes, SIMD machines offer synchronized execution. Without having to coordinate multiple synchronization points over tens of thousands of processors, the CM-2 is relatively simple to program. Machines like the CM-2 may not, however, perform well on array accesses, `if-then-else` statements and loops of differing iteration lengths. This is regrettable since non-numeric algorithms tend to contain these programming structures in abundance.

With the gracious support of the National Science Foundation, we have recently purchased a CM-5 from Thinking Machines. This machine is intended to combine many of the best features of the MIMD and SIMD models, and is expected to be a fertile testbed for future experimental work along the lines reported here.

Acknowledgment

We wish to thank an anonymous referee, whose insightful comments on the idiosyncrasies of the CM-2 architecture and whose careful critique of the original version of this paper have helped us to streamline and clarify this presentation.

nor as likely to depend solely on local operations. For this reason, non-numeric algorithms and their implementations exhibit a more balanced usage of computing resources and are well suited for studies of how the limited resources of actual machines may be utilized. The transformation of an algorithm from the Sequent Symmetry S81 to the Connection Machine CM-2 has helped illustrate that performance depends heavily on how these resources are handled by each machine.

6.1 Some Lessons Learned

There is much to be said for and against each of the architectures we have studied in this paper. In particular:

- Large-scale data movement is hampered by the single data bus design of the S81 while the router network of the CM-2 is able to move data between different processors without much apparent difficulty.
- Memory management on the CM-2 can be primitive, despite other advanced architectural features of the machine.
- Explicit user-placed synchronizations needed in the MIMD model and the inability to process different parallel tasks under the SIMD model are major drawbacks for complex non-numeric algorithms such as ours.

6.2 Architectural Performance

The wealth of built-in functions that are designed with efficient data movement in mind are a large advantage of the CM-2 over the S81. The `scan` function and it's many incarnations alleviate a large burden from the programmer and allow a somewhat higher level of programming to be done on the CM-2. Having to emulate some of these same functions on the S81 led us to appreciate the amount of effort that was saved through their use.

5.4 Task Synchronizations

Because of the limited types of S81 runtime data available, the precise amount of time taken by barrier synchronization cannot easily be determined. The best estimate that we were able to find was by running the merge code with only the `m_sync` calls intact. The number of loop iterations for data movement steps were approximated based on the size of data and number of processors being used. Times from these experiments ranged from 10% to 25% of the total execution times for the full merge with corresponding amounts of data and corresponding numbers of processors. This method of timing is not able to take into account the actual overhead time that processor interactions inflict on execution time. Thus, we assume that the sync times can be much higher in practice, especially as more processors are added.

While the execution time on a fixed number of processors doubled as the merge data size was doubled, doubling the number of processors for a fixed data set size did not cut the execution time in half. The impact of synchronization overhead is most apparent when comparing the execution times of steps 4 and 5 from Table 2. Both execute approximately the same number of data moves, yet the fifth step executes in 1/3 the time of the fourth step. There is no synchronization required in the local merge, where the data movement between processors in the fourth step is heavily synchronized.

Although the CM-2 does not require explicit processor synchronization, overhead is charged to the user for the SIMD lockstep mode of operation. There appears to be no accurate way of determining such overhead costs.

6 Conclusions

Most widely-studied parallel numeric computations have well-ordered patterns of memory access and processor cooperation. The performance of numeric codes tends to depend to a great extent on local operations. Non-numeric algorithms are generally not as predictable in effect 4 separate merges that must be executed in the final step of the algorithm with no overlap possible.

When the data sets are very small, there is a good chance that all processors will contain a short list. If there are no processors holding two lists of sufficient size, the full in-place merge is not executed—rather a buffer merge is performed—and the total execution time is kept small. As the total number of items grows, the likelihood of all processors having small lists decreases while there is still a good chance that at least one processor will have a short list (especially those processors that contain breakers).

Two other inherent SIMD serializations occur when using the C* `where...else` construct and arrays. The execution of the `where` clause is followed by the execution of the `else` clause by the appropriate processors. Again, no overlapping of separate executions is possible.

Access to array elements with a variable index is serialized by the front end processor. This is a critical architectural concern. A master index is iterated through all possible array index values and only those processors that have a matching index value are allowed to execute. Thus, in principle, an operation that appears to be $O(1)$ is really $O(n)$.

This is where the local merge time really suffers on the CM-2. An obvious way to try to reduce this time is with the use of local arrays at each processor. Indeed, implementations of simple not-in-place merge and sort routines using local arrays prove to be an order of magnitude faster at this step (as long as enough local memory is available). The use of local arrays will not generalize to the parallel algorithm as a whole, however, because memory access errors develop (for example, during the displacement table building step). Only specialized array handling libraries can correct this fundamental problem. Unfortunately, such libraries are not widely distributed for general use; accordingly, we did not use them in this study.

all 16 processors on the board into the same memory. By eliminating all but one processor performing the local merge, this last step of the algorithm exhibited a 20% average decrease in execution time from previous timings with all 16 PEs. The more than doubling of execution time when doubling the data size remained, however.

After conducting many separate experiments to trace the cause of this problem, it turns out that it is based on memory access on the CM-2. (One such experiment had each processor loop through n/k iterations performing a swap of two local locations from a large parallel variable. As the size of the parallel variable doubled, the runtime of this simple code also increased fourfold or more.)

Several consultations with experts at Thinking Machines uncovered that access patterns are highly non-optimal. We were advised that our only alternative was to augment the C* code with memory access functions available in CM-2 assembler (PARIS). Given our focus, we of course avoided such a low-level fix. Observe that this is exactly the sort of result we sought to uncover: the state of the art is still that one cannot reasonably expect straightforward, high-level transformation from one architecture to another without big disappointments in performance.

5.3 Single Instruction Limitations

As mentioned earlier, the final step comprises two distinct merge phases due to the duplicate nature of the data held in processors that contain breaker records. The SIMD paradigm requires that these two merges be executed in serial, one following the other across all the processors with breaker records. (Those processors without breakers remain idle during the execution of the second merge.) Additionally, within the local merge itself, there are two distinct merge routines: one for those processors that contain a very small list to be merged and the in-place merge for all others. Due to the lockstep execution mode of the CM-2, these must be run in a serial across all the processors participating in the merge. Thus, there are be moved between processors that are not necessarily located in close vicinity to each other. Folk wisdom holds that communication should be kept to an absolute minimum, and should be done over the smallest proximity of processors whenever possible. Nevertheless, the CM-2 routing network was able to shuffle large amounts of data between processors without much danger of running into the collision possibilities inherent in the single data bus on the S81. Moreover, the relatively low number of keys assigned per processor significantly reduces the number of data transfer instructions that need to be executed. (The largest data ratio achievable on the CM-2 was 128 keys per processor while the largest ratio on the S81 was 524288 keys per processor.)

On the S81, bus and memory contention can be reduced by performing the local merge on data held in the cache. The three intermediate steps of the algorithm require data transfer and cooperation between processors that are within the same series pair. Since data located between breaker records will remain between those breaker positions after the final merge completes, the more breakers contained in the data the tighter the data locality will be within these three steps. The large number of synchronizations required in the fourth step contribute to the large amount of time spent there.

The CM-2 performs well in these intermediate steps by being able to perform intrachip communication. Since processors are grouped 16 to a node, when the number of breaker records is very high, much of the required data transfers can take place between processors on the same chip and the same memory module.

One surprise in all this was the small percentage of time the Symmetry required for the local merge. In this step, all data can be manipulated within a processor's cache. As the algorithm rolls through the data, the in-place merge has a very predictable memory access pattern that makes for very high cache hit ratios. It was similarly surprising that the local merge took the most time on the CM-2. One explanation put forth was the serialization imposed by the SIMD model (more on this later). Another possibility was the contention of users. Dramatically better timing results would have been possible if certain sections of our algorithms were microcoded (as was done, for example, in [3]). But this was not our goal. We sought instead to identify algorithmic bottlenecks of the transformation process.

More interesting than any sort of a direct comparison of execution times between the two machines is a comparison of the strengths and weaknesses of both models by how each performed on the various steps of the algorithm.

On the S81, the execution times roughly double as the amount of data doubles for a fixed number of processors, while the execution times do not decrease by half when the number of processors is doubled and the input size is fixed. This suggests that certain features of this machine (discussed in § 5.4) do not scale very well with the addition of extra processors for algorithms such as this.

The parallel merge has quite the opposite effect on the CM-2. Twice as many processors take half as long on the same data sets. Yet, when the number of processors is held constant, a doubling of the data resulted in a quadrupling or more of the execution time. Uncovering causes of this behavior (discussed in § 5.2) has been illuminating.

5.2 Data Locality and Cache Effects

Data locality is often claimed to be one of the most important factors in (sequential and) parallel computations. Indeed, this is seen quite well in the step timings (Table 2) on the S81. The first and fourth steps move large numbers of keys from all over memory to different memory locations. This can swamp the single data bus. The processor cache is not able to alleviate this problem even though the movement of data is done in an orderly, predictable manner that could be made use of in a cache prefetch scheme. Since it has little or no effect on the execution of this step the loading of the cache may slow down these portions of the code.

One might assume that the CM-2 would also suffer from this problem since data must

Task	Execution Time (in seconds)	Percent of total time
Step 1 (Sort by tails)	1.81	2
Step 2 (Locate breakers)	0.07	0
Step 3 (Distribution Table computation)	1.85	3
Step 4 (Block rotation and Data movement)	6.35	8
Step 5 (Local merge)	63.87	86
Overhead (function calls, etc.)	0.46	1
Total	74.41	100

Table 4: SAMPLE CM-2 STEP TIMES

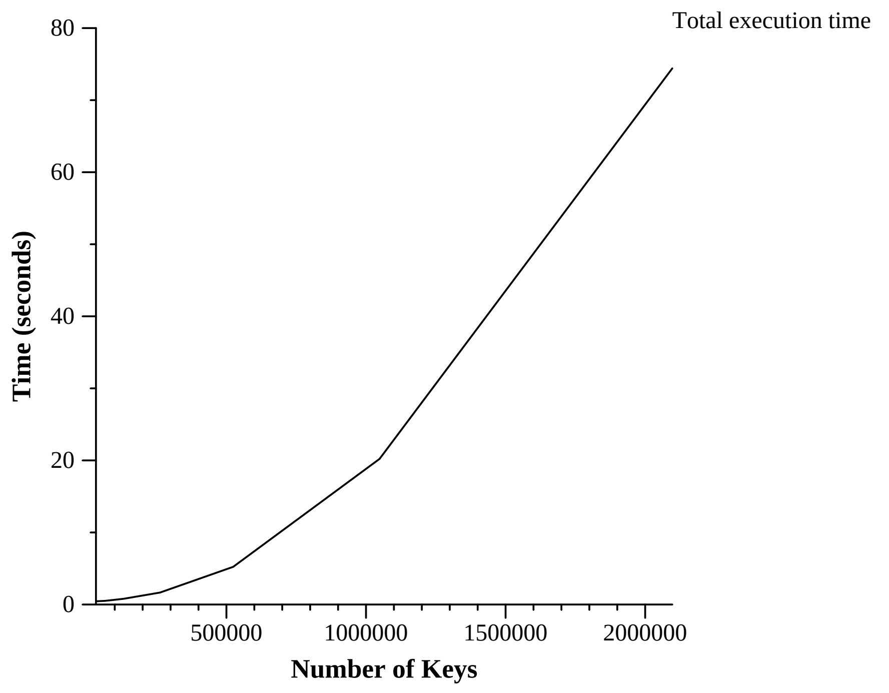

Figure 2: CM-2 TIMING RESULTS (16384 PROCESSORS)

Number of	Total number of keys merged						
processors	32768	65536	131072	262144	524288	1048576	2097152
16384	0.45	0.51	0.79	1.66	5.23	20.21	74.41

Table 3: Total CM-2 Execution Time

processors.

With 16384 processors, a maximum of 2^{21} (2,097,152) integer (4 byte) keys were allowed, matching nicely with the S81 limit. This bound gives a ratio of 128 keys per processor, and was the upper limit on this ratio no matter how many processors were employed. (In other experiments it was found that programs with fewer statements could compute with larger data sets.)

Table 3 lists the total amount of execution time (in seconds) spent on the parallel merge using 16384 processors with differing data set sizes. The results shown are based on the average of five random data sets and were generated on a 16384 processor CM-2 making full utilization of the machine and its connected resources. These results were obtained through the standard CM-2 timing routines of `CM_start_timer(1)` and `CM_stop_timer(1)`. Real time, CM time, front end virtual time (displayed here) and percentage of CM and front end utilization are reported by these routines. Table 4 shows the time taken by each individual step of the computations from merging 2097152 keys with 16384 processors. Figure 2 gives a graphical representation of the CM-2 execution times.

5 Interpretation of Results

5.1 Initial Remarks

Conversions from one architecture to another are unattractive if one must hack with low-level tools. Thus all our methods were implemented in high-level languages accessible to all parallel variable. The same broadcast can be done in a downward direction by placing large values instead of zeroes and performing a MIN scan. C* is designed to provide this type of segmented scan without having to rely on the values of the data to delimit the segments. Unfortunately, this feature was not implemented on the CM-2 we had available for our tests.

The use of a parallel scan as a broadcasting function led to an improvement in how the broadcasting of data could be done in the S81 code. Rather than the complicated algorithm originally proposed in [4], a form of parallel prefix and segmented scan was implemented. This approach sped up processing of the broadcast portions of the code by about 10%. Although this savings is insignificant in the overall execution time, the resultant code is considerably simpler.

Another form of `scan` operation are the reduction operators in C*. These operators reduce a parallel variable to a scalar value by combining the values of the parallel variable according to the given operation. Summation and bitwise logical operations, along with MIN and MAX are among the possible combining operations of the reduction operators.

The sorting of tail keys in the first step of the merge algorithm was performed by another specialized function, `rank`. The `rank` function returns a parallel variable of the same size as the given parallel variable. This returned variable holds the rank of the data values in the corresponding positions. This variable is then used as an index to swap the blocks.

4.3 Computational Experience on the CM-2

The CM-2 was not as flexible in allowing variation in run-time parameters. Once the size of the data was determined, the number of processors was the only other parameter that could be adjusted. The choices allowed on the test hardware were 8192 or 16384. Irrespective of the number of processors, we discovered that the timing results were remarkably well correlated to the number of keys allocated per processor. That is, the execution times for merging 32768 keys on 8192 processors was the same as merging twice as many on 16384 variables used in C*. Since each element of a parallel variable is thought to be assigned to a different processor, accessing elements that are not held by the requesting processor must be indexed. This indexing is placed before the variable name to distinguish it from the usual array indexing. (This left-indexing is executed as an indirect memory addressing into the shared memory, not an array indexing operation.) The `where` statement is the C* equivalent of a parallel `if-then-else`. Only those processors that evaluate the conditional statement as `true` are allowed to execute the `where` clause statements. Following this, if there is an `else` clause, those processors that had been idle for the `where` clause are then activated to execute.

The C* function library includes routines that can perform computation on parallel variables and transmit data at the same time. The `scan` operation is the most generic of such operations and is easily adapted to perform other, similar tasks. Parallel prefix operations are computed with the combination of the `scan` operator and the specification of how to combine elements of the parallel variable. Possible combiner functions are MIN, ADD, MULTIPLY and logical AND. The direction of a scan operation is determined by the user. The *upward* direction scans from the lowest processor index to highest and the *downward* direction is from highest to lowest.

The broadcasting of breaker values in the second step of the merge was performed by the scan facilities on the CM-2. To accomplish a scan that would only broadcast values from a single processor to all processors in the series, but not to others outside the series, each broadcasting processor assigned the broadcast value to the appropriate parallel variable. (The broadcast values were all local indices and were monotonically increasing as were the processor identification numbers.) All other processors placed a zero value in this same parallel variable. The scan then performed a MAX scan in the upward direction. Since the values to be broadcast were greater than zero, they replaced all the zero entries. The broadcast segment would be halted when the next non-zero element was reached within the each new iteration. Similarly, any loop structure that could execute on different processors with differing iteration counts had to be handled in a similar way. Crafting a method for simulating different length loops within the SIMD framework was straightforward.

Although each processor in the CM-2 must execute the same instruction that all other processors are executing, it is possible to specify that only selected processors execute statements while all others remain idle. A boolean parallel variable was declared and each element used to designate whether or not the loop on the corresponding processor had finished its iterations. Only those processors that had not finished were chosen to continue executing the bodies of the loops. After every iteration, each active processor would check to see if it had completed the loop executions. If it had, the flag was reset to `false` and the processor would become idle. When all the processors flags were `false`, every processor would proceed to the next program step. The checking of all flags was performed using a boolean OR reduction operator (`|=`) illustrated below.

For example, the following MIMD parallel code

```
i = j;
while (a[i] <= temp) {
  if (b[k] > a[i]) k--;
  i++;
}
```

would be coded as

```
i = j;
more = ([i]a <= temp);
while (|= more)
  where (more) {
    if ([k]b > [i]a) k--;
    i++;
    more = ([i]a <= temp);
  }
```

on the CM-2 where `more` is the boolean parallel variable used to indicate which processors have more iterations to complete. Note the reversal of the variable-index order for parallel data access patterns and length of messages, is typically 80 million to 250 million accesses per second on a fully configured system.

4.2 Code Conversion

C*, the Thinking Machines data parallel version of the C language, was used to implement the parallel merge on the CM-2. In C*, data spread across the CM processors are placed in *parallel variables*. Each parallel variable has a *shape* that is determined by the dimensionality of the data and the number of elements in each dimension. Though variables in more than one shape can be declared, a C* program generally works with only one shape at a time (the *current* shape). To operate on a particular parallel variable, the shape of that variable must be made the current shape. The parallel variable holding the data to be merged was one dimensional, and larger than the number of processors at hand. (If the current shape holds more data elements than processors are available, then multiple virtual processors are simulated at each real processor.) Additional local variables to be used in each processor were declared to be of shape *physical*, which is the intrinsic shape that allocates a single element of the parallel variable to each physical processor. By using these variables as the current shape, we were able to implement the algorithm without using virtual processors.

The major problem in coding the MIMD implementation was the number of synchronization points that had to be included in the code and the coordination of the multiple tasks in executing the same number of barrier synchronizations at the proper times. With a SIMD machine, the synchronizations are done automatically. While this makes coding of processes that are tightly coordinated much simpler, it does impose some restrictions on the capabilities of the machine that must be overcome.

For instance, the C* language does not support parallel `for`-loops. In the conversion of the code from the S81 to the CM-2, all such loops needed to be "hand coded" by initializing the counter, incrementing it and checking to see if the exit conditions had been reached before

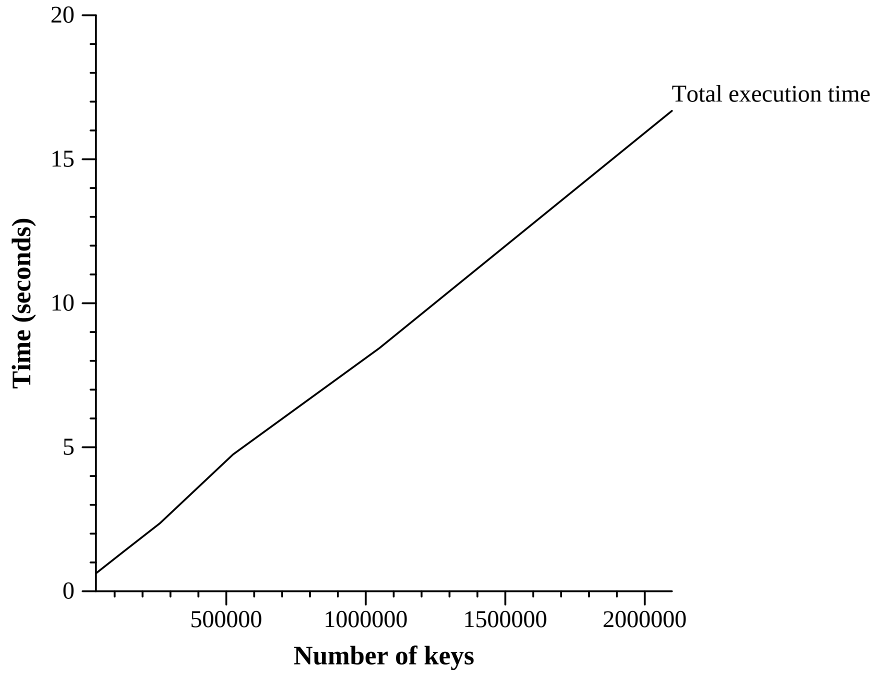

Figure 1: S81 Timing Results (16 processors)

4 Extension to the SIMD Model

4.1 The Connection Machine CM-2

The CM-2 is a data parallel computer. Systems may contain as many as 65536 bit-serial processors each with 64 Kbits of local bit-addressable memory. Groups of 16 processors are contained within a single chip as a node that is in turn connected to other nodes in the system through a hypercube configured router network. A standard serial computer acts as a front end processor executing the serial portions of the user's code and controlling the execution of the parallel code on the CM-2.

The bit-serial processors are proprietary to Thinking Machines, Corporation. CM-2 models may also be equipped with floating point chips connected to groups of 32 processors. The data communication router network on the CM-2 is used for general, point-to-point processor communication. The data throughput rate on the router network, though depending on

Number of	Total number of keys merged						
processors	32768	65536	131072	262144	524288	1048576	2097152
2	1.11	2.23	4.16	8.78	17.59	35.43	71.75
4	0.76	1.40	2.80	5.30	11.13	20.63	40.26
8	0.54	0.94	1.67	3.25	6.25	12.68	24.58
16	0.62	0.87	1.37	2.36	4.75	8.44	16.68

Table 1: Total S81 Execution Time

Task	Execution Time (in seconds)	Percent of total time
Step 1 (Sort by tails)	4.49	27
Step 2 (Locate breakers)	0.00	0
Step 3 (Distribution Table computation)	1.04	6
Step 4 (Block rotation and Data movement)	7.77	47
Step 5 (Local merge)	2.47	15
Overhead (function calls, etc.)	0.91	5
Total	16.68	100

Table 2: Sample S81 Step Times

microsecond counter `usclk` was used to time the test runs.

Table 2 shows the average amount of time spent on each step using 16 processors with 2097152 elements to be merged. This timing was done to determine how each step was performing in relation to the others. Those steps that were taking the most time could then be looked at more closely for bottlenecks and coding inefficiencies.

The graph in Figure 1 shows the relation of the overall total execution time on a differing number of keys on 16 processors.

The parallel merge exhibited very predictable behavior on the S81. A doubling of the data size resulted in a doubling of the execution time for a fixed number of processors. While we were satisfied with this result overall, we were somewhat surprised at the low efficiency synchronization forces on the computation.

This is very often not the case, however, as our debugging experiences have made clear. For example, without the second `m_sync`, other S81 users may preempt some but not all executions of the assignment to `num` so that the value `num` receives in the delayed processors is incorrect.

The full potential of the machine's MIMD nature is realized in the local merging phase. Once all the local pointers have been computed, each processor can perform its own separate in-place merge on data that will not be accessed by other processors. Because processors that contain breakers must execute up to two local merges, one final `m_sync` point is used to ensure all processors have finished before the merge is completed.

Our experimentations brought forth some unexpected results. We illustrate with the following anecdote. Initial runs of the Sequent code revealed that the parallel version of the merge was curiously slow, even slower than the sequential version no matter how many processors were used. Since $O(1)$ extra storage per processor is sufficient in theory, the preliminary coding had employed only one extra memory location to be used for swapping. But six is also $O(1)$, and with just a half dozen extra swapping locations and 1/6 the number of synchronizations it became routine to get parallel execution times that were respectable.

3.3 Computational Experience on the S81

The architecture and programming of the Sequent Symmetry allowed for many different variations in parameter values under which to test the parallel merge algorithm. A maximum of 2^{21} (2,097,152) integer (4 byte) keys could be merged.

Table 1 shows the total amount of time spent (in seconds) by the parallel merge for a select number of processors over a number of data sizes. The times listed include the time for synchronizations (which are frequently ignored in the literature) and represent average times taken from five different randomly generated data sets. Outside activity on the target machine was non-existent or at a minimum when the test runs were timed. The 32-bit point in the code. After the processors have executed the barrier synchronization command, all processors are allowed to proceed with their next instruction. The coordination of the synchronizations can be a chore in itself for the programmer. Since each call to m_sync is identical to every other call, processes can actually be synchronized at different points in the program. For example, after the blocks are sorted by their tails and the breakers are found among the blocks, any blocks that are situated after the final breaker are already in sorted order and the processors that are handling those blocks need not participate in any following execution. Once these processors are identified, it is an easy matter to ensure that they do not execute any further code that may move the records they are holding and consequently unsort them. However, because the remaining active processors must cooperate among themselves, there still remains a number of synchronization points that must be met and passed. If reciprocal m_sync instructions are not programmed in for those processors that have finished processing early, the synchronization of the whole program will be compromised and the system can easily become deadlocked. Adding processors only magnifies the difficulty of tracking down and debugging this kind of error.

Since calls to m_sync involve a function call and its attendant overhead, it is tempting to omit a synchronization point to reduce execution time, especially when you have code of the form

```
m_sync();
num = blanks[myid-1];
m_sync();
if (temp != key[i])  blanks[myid]--;
```

where num is local to the processor, blanks is a shared variable and myid holds the local processor identification number. Since the last statement of this code modifies blanks[myid], the second m_sync is necessary even though only one statement is executed after the previous m_sync. A user might expect that a single statement should be executed by all processors in the same amount of time, especially one that directly follows a synchronization point.

3.2 Implementation Details

Having been designed with the shared-memory MIMD model in mind, the implementation of the aforementioned parallel merge onto the S81 was not difficult. The C language was chosen, augmented with Sequent Parallel Programming Library routines needed to perform communication and synchronization.

Two categories of variables can be used: *shared* or *private*. The data to be merged and the auxiliary variables whose values are to be used by more than a single processor are declared to be shared. All others are private and treated as if they were local to each processor. Each processor knows its own identification number in relation to all other processors; consequently, the indices for the boundaries on the blocks of the files that are under any one processor's control are easily calculated.

PRAM models typically assume a lockstep execution of instructions ([1, 6, 7]). In real MIMD machines, no such lockstep is usually enforced. Each processor is given its assignment and allowed to run the instructions asynchronously. Should a sharing of data or other cooperation between tasks on separate processors be necessary, the program must provide some method of synchronization between the cooperating processors. For the parallel merge implemented on the Sequent Symmetry there is a large amount of cooperation required among processors within the first four steps of the algorithm. Many portions of the code require processors to use the values of pointers and counters held by a neighboring processor. Nearly every one of these types of access must be preceded and/or followed by a global barrier synchronization to ensure that processors do not violate EREW restrictions. The choice of barrier synchronization over other methods is based on availability and programming simplicity. Subsequent coding of portions of the merge algorithm using locks in place of barriers resulted in much slower execution times.

The barrier synchronization `m_sync` is used to synchronize all processors to a single

The third step computes the number of records in each first series block that would be displaced by records from the second series if there were no other first series blocks. This value is held in a *displacement table*. Block i can use these table values to determine exactly how many records will be displaced by records from block $i-1$ and how many will actually be displaced by records from the second series.

The fourth step distributes the records of the second series among the blocks of the first series as well as shifting displaced records between the first series blocks. This step is accomplished through a succession of local block rotations followed by appropriate parallel data movements between processors.

The fifth and final step (local merging) is implemented as a choice between two time-space optimal merges [5] dependent on the relative sizes of the two subfiles held by the processor. Processors with breaker records may need to perform separate merges for the portions of data located on either side of the breaker's position.

3 An MIMD Implementation

3.1 The Sequent Symmetry

The Sequent Symmetry is a tightly-coupled multiprocessor with a shared common memory. The CPUs, memory modules and I/O controllers are all attached to a single high-speed bus. A maximum of 30 processors can be incorporated in a Model S81 system.

The configuration we tested, located at Argonne National Labs, employs Intel 80386 processors with Intel 80387 and Weitek WTL 3167 floating point coprocessors. System memory contains 32 Mbytes; each processor is equipped with 64 Kbytes of cache. The 64-bit system bus has a channel bandwidth of 80 Mbytes per second. The actual data transfer rate achievable is 53.5 Mbytes per second.

2 A Review of Time-Space Optimal Parallel Merging

For the reader's benefit, we now briefly outline the time-space optimal parallel merge we have implemented. The method we describe is robust, ensuring time-space optimality even on the weak EREW PRAM model. It can be used in time-space optimal sorting as well with a straightforward sort-by-merging approach. We refer the reader to [4] for a complete description of these techniques.

In what follows, we assume a file with a total of n records is input containing two sorted subfiles to be merged. We let k denote the number of processors available. For simplicity, we assume each of the two sublists is evenly divisible by k.

The parallel merge comprises five steps. The first four reduce the problem to one of local merges at each processor (the last step). The data to be merged is viewed as k blocks, each block of size n/k, and each block managed by a distinct processor.

The first step is to sort the blocks by their tails (largest keyed records). This is accomplished by extracting a copy of each tail and its corresponding processor index, and sorting these with a bitonic merge [2]. Blocks are moved to the appropriate memory locations using the processor indices that were sorted along with the tail values.

The second step divides the data into pairs of series. Boundaries of the series pairs are located at *breakers* (such a breaker is the first record in block $i+1$ whose key is no smaller than that of the tail of block i). The first series of a pair contain records that follow a breaker and are all from the same original sublist. The second series in the pair will end with the record just before the next breaker. When a processor determines that it has a breaker in its block, the index of the breaker is broadcast to the other processors. Thus, each processor can determine the boundaries of the series pairs around it and whether or not the block it holds is in the first or second series of a pair. It should be noted that processors that contain breaker records will have both a second series and a first series in their block of data.

1 Introduction

The purpose of this paper is to study design and implementation issues for non-numeric parallel algorithms on real MIMD and SIMD machines. As opposed to floating-point intensive codes, non-numeric algorithms are fairly unpredictable with respect to memory accesses, non-local data requirements and processor cooperation. We center our investigation on the merge and sort algorithms from [4]. These methods are attractive because of their scalability with respect to both time and space. That is, they ensure *time-space optimality*, so that optimal speedup is attained[1] and yet they require only a constant amount of extra space even when the number of processors is fixed.

Our representative MIMD and SIMD machines are the Sequent Symmetry S81 and the Connection Machine CM-2, respectively. The S81 employs a handful of fairly powerful processors; the CM-2 uses tens of thousands of relatively simple ones. The S81 connects its processors to a shared memory through a common bus; the CM-2 connects its processors in a hybrid binary hypercube fashion with memory distributed among the individual processors.

To our knowledge there has been no previously published case study describing the conversion of shared-memory parallel codes to the data parallel paradigm. We develop techniques that have been helpful for us and that may be useful in future conversions of this nature. In addition to our discussion of implementation issues, we present sample timing results from both machines. We emphasize that our goal is not to fine-tune our codes to make them competitive with microcode implementations available elsewhere, but rather to discover stumbling blocks inherent in the MIMD-to-SIMD program transformation process. Thus these timing figures, slow though they may be, are useful in identifying the relative strengths and weaknesses of these two divergent models.

[1]A parallel method attains asymptotically optimal speedup if the product of the number of processors it employs and the amount of time it takes is within a constant factor of the time required by a fastest sequential algorithm.

To Appear in *Journal of Parallel Algorithms and Applications*

MIMD Versus SIMD Computation: Experience with Non-Numeric Parallel Algorithms[*†]

Clay P. Breshears and Michael A. Langston

Department of Computer Science
University of Tennessee
Knoxville, TN 37996-1301

Abstract

We focus on differences inherent in the design and implementation of non-numeric parallel algorithms on MIMD and SIMD architectures. We take as our prototypical examples time-space optimal merging and sorting routines. Our representative MIMD and SIMD machines are the Sequent Symmetry S81 and the Connection Machine CM-2, respectively. In addition to the contrast provided by their differing execution philosophies, this choice of machines allows us to compare results from both shared-memory and distributed-memory models.

Unlike many numerical parallel algorithms, non-numeric parallel programs are rarely well structured with respect to inter-processor cooperation, memory access and communication patterns. This is particularly apparent when one must deal with delicate issues such as process synchronization in the MIMD model and coding variations in the SIMD model.

Our experiences are highlighted with examples obtained during the process of MIMD to SIMD program transformation. Unexpected events can occur when algorithms designed with one architectural style in mind are modified for execution on another. Timings and other measurements are useful in identifying important bottlenecks. We discuss some relative strengths and weaknesses of the two competing models that have become evident during this transformation process.

[*] A preliminary version of this paper was presented at the 26th Hawaii International Conference on System Sciences, held on the island of Maui, Hawaii, in January, 1993.

[†] This research has been supported in part by the National Science Foundation under grant MIP–8919312 and by the Office of Naval Research under contract N00014–90–J–1855.