Dynamic Parallelization for RNA Structure Comparison

Eric Snow, Eric Aubanel, and Patricia Evans
Faculty of Computer Science
University of New Brunswick
Fredericton, New Brunswick, Canada E3B 5A3
Email: {eric.snow, aubanel, pevans} @unb.ca

Abstract

In this paper we describe the parallelization of a dynamic programming algorithm used to find common RNA secondary structures including pseudoknots and similar structures. The sequential algorithm is recursive and uses memoization and data-driven selective allocation of the tables, in order to cope with the high space and time demands. These features, in addition to the irregular nature of the data access pattern, present particular challenges to parallelization. We present a new manager-worker approach, where workers are responsible for task creation and the manager's sole responsibility is overseeing load balancing. Special considerations are given to the management of distributed, dynamic task creation and data structures, along with general inter-process communication and load balancing on a heterogeneous computational platform. Experimental results show a modest level of speedup with a highly-scalable level of memory usage, allowing the comparison of much longer RNA molecules than is possible in the sequential implementation.

1. Introduction

Dynamic programming algorithms, known for both their high complexity and high computational demands, have the potential to benefit greatly from the increase in resources available from parallel computers. However, a considerable amount of thought must be given to the design of the parallel versions of such algorithms, as communication overhead caused by intermediate result dependencies can potentially result is designs that cannot run efficiently in practice [7].

Parallelization of iterative implementations of dynamic programming algorithms with high space and time complexities may not be sufficient to allow the solution of large problems. In [6], Evans presented an algorithm to find common RNA pseudoknot structures in polynomial time. Given this algorithm's worst case $O(n^{10})$ time and $O(n^8)$ space complexities, a recursive implementation was presented, using memoization and data-driven selective allocation of the tables, which dramatically reduced the actual space requirement and running time. In this paper we consider the parallelization of this implementation, since parallelization of an iterative implementation of the algorithm would not be feasible.

There has been significant recent work on the parallelization of dynamic programming algorithms in computational biology [9], [10], [11], including implementations suitable for computational grids [5]. What distinguishes this work is the data-driven recursive implementation, with resulting dynamically allocated tasks.

The rest of this paper is organized as follows. In section 2, we introduce the sequential version of the algorithm and discuss its qualities with respect to how they affect parallelization. Section 3 provides information regarding the parallel algorithm design and implementation. The evaluation of the implementation is shown in section 4, and conclusions about the completed work are discussed in section 5.

2. Finding Common RNA Pseudoknot Structures in Polynomial Time

While sequence comparison in RNA has become a significant area of study in recent years [8], additional information can be gleaned from the comparison of secondary structures formed by the hydrogen bonds between pairs of bases. These structures may not hold significant sequence similarity but can impact functionality, making locating common structures within a pair of RNA strands an area worthy of study. In particular, finding the maximum common ordered substructure (*MCOS*) between a pair of RNA strands, while maintaining both the strands' sequences and the ordering of the bonds has been an area of focus [2], [6].

2.1. Sequential Algorithm

While finding the *MCOS* for an arbitrary set of structures of length n and m (where $n \geq m$) has been shown to be NP-complete [12], an algorithm with worst-case time and space complexity of $O(n^4)$ was created by disallowing bonds which share endpoints and bonds which cross, while continuing to allow bonds which precede one another or nest within one another [2], as is illustrated below in Fig. 1.

While this algorithm is very efficient for finding the *MCOS* of certain types of RNA, it disqualifies the com-

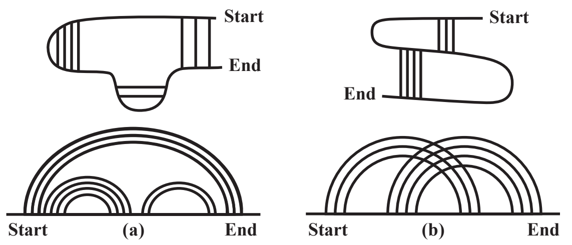

Figure 1. Illustration of bond structures with (a) nested and preceding bonds and (b) crossing (knotted) bonds.

parison of many real-world RNA structures in which certain types of knotting (including pseudoknots and pseudoknot-like structures) do exist. By allowing only types of knotting consistent with actual RNA, a dynamic programming algorithm which we refer to as the *Pseudoknot Algorithm* was developed in [6] which has worst-case time and space complexities of $O(n^{10})$ and $O(n^8)$, respectively.

2.2. Sequential Implementation

Based on the significant amount of recurrence and call repetition, the pseudoknot algorithm is implemented as a recursive dynamic programming algorithm using memoization, with 4- and 8-dimensional tables used to store intermediate results. Due to the extreme time and space requirements of calculating all intermediate points in the 8-dimensional table for even small-sized input structures, a selective computation and allocation approach is used. The algorithm allocates slices in the tables and performs calculations only when they correspond to cases consistent with the input RNA structures. The implementation was shown to use an average of only 10^{-14} of the worst-case space, supporting input structures of up to 400 bonds in 4Gb of memory [6].

The motivation behind the parallelization of the algorithm is twofold. First, while the implementation is able to avoid the the pitfall of being impractical due to its worst case time and space complexities, the amount of time and more importantly, space required does still increase at a high rate as the size and complexity of input structures increases. Any attempt at work with such structures would be difficult without the large amounts of main memory available through parallel machines. Additionally, the sequential algorithm and implementation have several qualities that make it interesting and unusual for a problem that needs parallelization.

2.3. Sequential Analysis

A *primitive task* in the sequential implementation, indicating the smallest piece of stand-alone work, is represented by a call to return a point in one of the dynamic programming tables. If the point already exists, a simple return of the stored result ends the task, while a full computation of the task's result must occur prior to returning if it has yet to be found. At the beginning of the algorithm, a single task is created, which will contain the solution to the problem when it completes. During the solution of a task, other tasks are created dynamically when the solution of a point in the table is needed.

The pseudoknot algorithm can be classified as a non-serial polyadic dynamic programming algorithm with irregular dependency. The *non-serial* classification refers to the fact that to find the solution to a given task in the recursion tree, one of the tasks needed to complete the solution may be more than one level away. The *polyadic* nature of the algorithm means that the solution of any task will require the results of *at least* two other tasks. Finally, the algorithm exhibits *irregular dependency* in that the solution of a task can require a variable number of solutions of other tasks. This classification makes the pseudoknot algorithm amongst the most difficult to parallelize efficiently, as the potential for high communication costs is significant. This ensures that a traditional static communication design will not be possible.

The major difficulty with the parallelization, including the irregular dependency, stems from the selective calculation and allocation used in the implementation. This technique, which allows the input data to drive computation, is invaluable to reducing memory and time requirements, far more than could be saved by a normal parallel implementation that did away with it to use a simpler data decomposition. As a result, the inherent parallelism in the problem is greatly diminished.

The selective nature of the implementation places an *order* on the calculations that would not be present otherwise. Because it is not possible to determine whether a given point is going to be valid prior to reaching it during execution, it is not possible to simply guess whether a task will be necessary without potentially adding a significant amount of unnecessary time- and space-consuming calculations. This leaves only one true starting point for calculation, ensuring that a *staggered* processor execution order is the only viable option.

The data structures worthy of concern are the 4- and 8-dimensional dynamic programming tables used to hold the results of tasks. These tables can grow to be extremely large, and due to selective allocation gain an *irregular shape*, with different indices of the same dimension containing different numbers of entries. This irregular shape is prevalent only in certain dimensions of each table. Specifically, the arrays making up the first and third dimensions of the 4-dimensional table are always constant sizes n and m respectively, as are the arrays making up the first and fifth dimensions of the 8-d table. The sizes of the arrays making up the other dimensions of each table are not known until that particular slice is needed.

3. Parallel Algorithm and Implementation

The parallelization of the pseudoknot algorithm uses a heterogeneous distributed-memory underlying platform with message passing capabilities as the basis for its design and implementation. Constrained by the fact that it must make use of the selective computation and allocation discussed previously, along with the need for the dynamic programming tables to be distributed to allow for more available memory, the design follows a new parallel programming paradigm known as *manager-worker*. Manager-worker is a highly modified version of the master-slave paradigm, sharing some similarities with the recently proposed *scheduler-worker* paradigm, which is designed for robustness under disturbance on heterogeneous platforms [5]. Where the manager-worker differs from the scheduler-worker is in the management of its tasks and task-related communication. The scheduler-worker paradigm was designed for parallel algorithms with a static number of tasks, allowing the scheduler to quickly redistribute the task allocation to every worker whenever a round of load balancing takes place. However, since task generation in the pseudoknot problem is extremely dynamic and unpredictable, the opportunity for one large schedule to be efficiently created and distributed to the workers is not feasible. Instead, based on periodic updates from the worker nodes, the manager has the opportunity to initiate load balancing procedures between two workers if needed, moving certain tasks and structural data between them. Fig. 2 shows an illustration of the manager-worker paradigm.

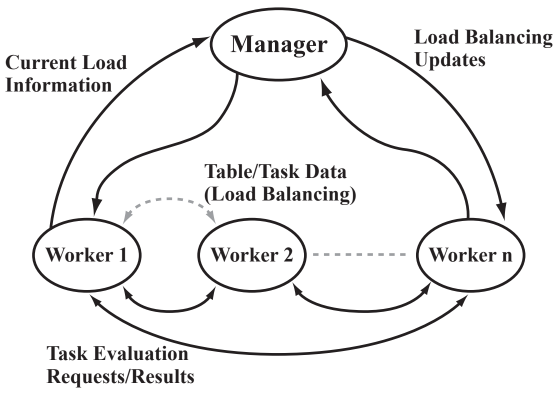

Figure 2. The manager-worker paradigm. Dashed arrows are blocking messages, while bold arrows show non-blocking messages.

The design of the manager processor's role is that of an overseer of the entire operation, without having to know which individual tasks are being created and carried out. This relegates the manager to the evaluator of whether load balancing must occur at any given time, a responsibility that is vital in a situation where both tasks and data structures are created dynamically. The probability of a skewed data distribution is quite high, and measures must be taken whenever necessary to ensure that workers have both enough work and enough space in which to do it.

The workers are responsible for task creation and evaluation, the creation and maintenance of dynamic programming tables, and the majority of communication that takes place during the course of the program. The tasks dynamically created during evaluation can require communication with other processors for results, and the load balancing procedures themselves involve only the worker processors. Pseudocode 1 shows a high-level view of the worker program flow; note that the portion of the 4-dimensional table located on processor 1 will always contain the final solution (assuming that the manager is processor 0).

Pseudocode 1 Worker Processor Functionality

```
findSolution():
  receive initial data structures from manager
  if processor id = 1
    while table4[$0, n, 0, m$] has not been solved
      if this processor's task list is empty or threshold tasks have
      been evaluated since the last communication phase
        enter communication phase
      else, evaluate the node on top of the task list
    send table4[$0, n, 0, m$] to manager
  else
    while no completion message has been received
      if this processor's task list is empty or threshold tasks have
      been evaluated since the last communication phase
        enter communication phase
      else, evaluate the node on top of the task list
```

The implementation begins with the first worker attempting to evaluate a single task, which will result in more tasks being created, some of which may require requests from other workers. The involvement of additional workers is *staggered* in an unpredictable manner, in that it is dependent on the input structure, and as such it can fall to the manager to use load balancing to get more workers involved. The manager, in addition to monitoring for uninvolved processors, also monitors for processors running out of available memory used for dynamic programming tables, and can take load balancing measures to remedy any issues.

3.1. Data Structures

The 4- and 8-dimensional dynamic programming tables are distributed amongst the worker processors at the beginning of execution. The tables are split along the first dimension's indices due to the fact that the length of the dimension is static in both tables, and it allows for a short, simple mapping array to be used to represent the distribution of the data structure on all processors.

A copy of this mapping array is held by each processor, allowing workers to quickly identify which processor a piece of data is located on in a single operation. Additionally, it allows a simplification of load balancing procedures, as the shifting of task and tabular data can be described simply by changes to the mapping array. It is assumed that each worker always holds a contiguous selection of indices to minimize the amount of inter-worker data requests. This also helps to simplify load balancing by only allowing processors to exchange indices with those directly before and after them.

The final wrinkle with respect to the data structures is the need for reallocation. In the sequential implementation, the allocation of each non-static dimension was final due to the strict order of task completion. However, in parallel, this order cannot be maintained and allocations can potentially occur out-of-order, leaving a need for re-allocation. While this reallocation is extremely rare, it still necessitates the need for *assistant* tables which are used to keep track of the lengths of each array the non-static dimensions, as is illustrated in Fig. 3.

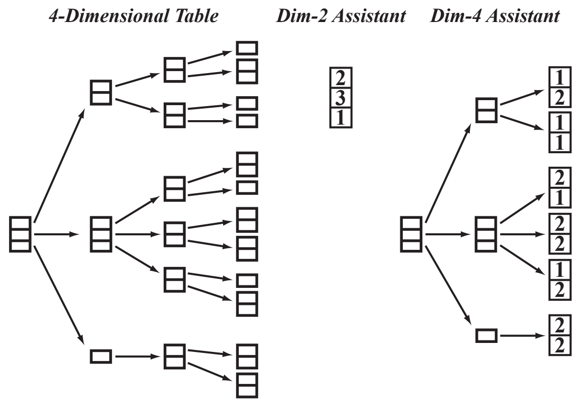

Figure 3. An illustration of the additional tables required by the 4-d table, used to track the lengths of the second and fourth dimensions

3.2. Task Creation, Evaluation, and Management

Each task in the parallelization is equivalent to a call to return a position in the 4- or 8-d table. The distributed nature of the tables in the parallel version adds a wrinkle to this, dividing tasks into groups, those requiring results from another worker and those whose results can be calculated on the current worker. A traditional use of recursion is not viable for handling such a situation, as massive portions of the recursion tree would need to be kept in memory while waiting on results from other workers.

To deal with the possible need for communication, a list acting as a modified stack is used on each worker to keep track of tasks. The task at the top of the list is evaluated, with any tasks generated during evaluation pushed to the top. To ensure that nodes waiting on results from other workers do not stall the list, they are appended to the rear instantly if they are at the top of the list and their communication has not yet returned. This setup ensures that the order-of-evalution of points in the table is as similar to the original sequential implementation as possible (reducing the number of reallocations needed), while allowing for flexibility needed by communication-based tasks. Pseudocode 2 shows a high-level view of the work that must take place when a call is made to evaluate a task.

Pseudocode 2 Worker processor evaluation of a task

```
evaluateNode(headNode):
    check the score in the dynamic programming table for headNode
    if the score has been successfully calculated
        if headNode is a request from another processor
            add headNode to the outgoing list
        else, pop the headNode and destroy it
    else
        if headNode is a duplicate task, append it to the rear of the list
        else
            for each point p in the table required to find
            the solution of headNode
                if the solution has not been found for p
                    if a node representing p has already been created,
                    but is waiting on communication
                        append headNode to the rear of the list
                    else
                        if p's location in the table is on another processor
                            create a node for p in the request list
                        else, create a node for p to add to the main list
                else
                    if p's solution is maximal
                        set the current score of headNode to p's solution
            if each point's solution was found
                enter the score of headNode into the table
                if headNode is a request from another processor
                    add headNode to the outgoing list
                else, pop and destroy headNode
            else, push all newly-created nodes onto the main list
```

3.3. Communication

Inter-processor communication is required to deal with two separate areas: task requests/results and load balancing. In order to simplify the process as much as possible, communication does not take place during general task evaluation but rather is confined to the *communication phase*, entered depending on the current state of the worker. This entry is triggered whenever a worker is starved or has gone above a threshold of attempted task evaluations since its last communication phase.

Load-balancing communication between manager and workers is kept relatively simple, with workers sending messages containing current load and list length information, and managers sending new mapping array information when load balancing occurs. These messages are passed in a non-blocking format and are short to prevent the manager from becoming a bottleneck.

Communication related to requesting task evaluation and the return of task results between workers, simply titled *task communication*, is extremely unpredictable in the sense that it is not possible for one worker to predict the phase of communication during which the result of its request will be returned. The number of calculations performed by another processor to find a certain point is not predictable, so it is possible for a request sent out during one communication phase to not be returned for a long period, while another request may be calculated and returned almost immediately.

To accommodate this, communication is designed to take advantage the fact that tasks do not need to be evaluated in an exact order. While the evaluation of a task does depend on an exact set of intermediate points from the dynamic programming tables, it is possible to move on to another task if the current one is waiting for a result from another worker. This means that workers need not wait for results to continue working, and allows for a flexible communication style that works well to alleviate some of the issues limiting parallelism.

Specifically, unless a worker's task list is empty or all its remaining tasks are waiting on communication, there is no need to wait during task communication. Task communication is performed in a non-blocking fashion, such that a worker can first send out any task communication that it has created since the last phase using non-blocking messages, and then receive any messages that have already arrived by probing for incoming messages using a non-blocking probe, exiting the communication phase immediately afterwards. Outgoing and incoming task communication can involve any other worker processor depending on the requests/results required, but messages will never be sent out to workers whose work is not required.

3.4. Load Balancing

Load balancing can be divided into two logical categories: the decision making process and the execution. The decision making process falls entirely to the manager, and is dependent upon the most recent load-related information from the workers, along with information describing the underlying platform. In addition, historical information detailing recent load balancing attempts on the processor are also considered, as due to the time overhead, repeated load balancing of the same processor can potentially cause it to become a bottleneck for other processors waiting on its results.

Load balancing is limited to one pair of workers per communication round, leaving the manager with the decision of which two, if any, are most in need. The load balancing score for each pair is calculated based on whether one of the pair have achieved the *critical level* based on the percentage of total memory in use. The critical level of a worker varies based on the underlying connectivity, recent load balancing involvement, and the potential amount of data being transferred, each of which have the potential to increase the critical level that must be achieved prior to a pair of workers being considered.

If one pair of workers does reach the critical level, it is assigned a score based on the relative severity of its current load balancing situation. At the end of the manager's evaluation, the pair of processors with the highest score (assuming there is one - often no pair will achieve the critical level), will begin load balancing. This design is set to encourage load balancing early in execution when it is not as costly, to allow many processors to begin execution as soon as possible.

When the decision to perform load balancing has been made, each worker must synchronize its communication prior to the execution of load balancing, including the completion of any ongoing *task communication*. Once this is completed, each worker not directly involved in the load balancing must update its list of tasks, such that any task currently waiting on a result from the processor losing its indices must be sent to the processor receiving the new indices during the next communication phase. After this is complete, the uninvolved workers can resume their regular task evaluation.

For the worker giving up its indices, tabular data must be put into buffers for transfer (one for each dimension) and the portions of the tables deallocated. One bonus of this technique is that by using a buffer for each dimension, the *assistant* tables used to aid in reallocation need not be transferred, as their dimensions and data can be gleaned from the buffers. The worker receiving indices can build the new portion of the table and the assistant tables in a once-through fashion once all buffers have been received.

4. Experimental Results

Coded in C using MPI, an implementation of the parallel algorithm has been created to test the functionality and efficiency of various aspects of the design. The testbed for experiments is the hybrid parallel cluster *Fundy* at the University of New Brunswick's ACEnet-sponsored high-performance computing facilities. Fundy is classified as a "big node" cluster, characterized by a large processor-to-node ratio [1]. The portion of Fundy used for the experiments runs Red Hat Enterprise Linux AS 4 (Update 4) and is composed of nodes containing 8 2.8 GHz dual-core AMD Opteron processors with 64 Gb of memory available per node, connected with Gigabit Ethernet. All experiments use 4 processors (cores) per node as their default distribution.

Test data for the experiments falls into two major groups: ribosomal RNA [4] and M1 RNA from Ribonclease P [3] and similar input structures. Input structures of the latter type is generally more complex and use significantly more space and time find the solution for an equivalently-sized input structure. Sample data for ribosomal RNA with up to 4000 bases was used, whereas M1 RNA data with up to 600 bases was used. To test the memory growth at higher values, artificial input structures similar to M1 RNA were created of up to 1250 bases as well. Only RNA containing pseudoknots and pseudoknot-like structures were considered.

4.1. Memory Usage

Unlike many other parallel implementations, speedup and efficiency are not the only, nor even the main, factors in determining whether the implementation can be declared a success. More than any other factor, proper memory usage and management is the motivation behind the parallelization of the pseudoknot algorithm. While a great deal of the time complexity is shaved off in the sequential version, the memory requirements still grow at a rate which is nearly impossible to manage on a single processor as the input size and complexity increases. Thus, the parallel implementation's main goal is to allow for larger and more complex input structures by distributing memory usage. Fig. 4 shows the average memory growth as the size of *n* increases.

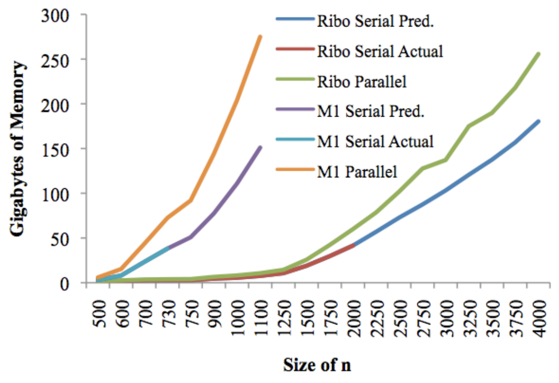

Figure 4. Average memory usage as the size of input increases

It is important to note that this is the average, as depending on the input structures the memory usage can vary a great deal. Estimates of sequential memory usage where total memory usage is > 40 Gb were calculated by the parallel implementation, as it was not possible to reserve more than 40 Gb for a single processor. Knowing this, the graph shows a significant number of new problem instances can be calculated using the parallel version which were not possible previously.

The sizes of the dynamic programming tables produced in the sequential and parallel implementations are identical because the positions calculated are identical. However, the use of the assistant tables, as illustrated previously in Fig. 3, is a pure parallel memory overhead which is not present in the sequential implementation. The amount of extra memory used to hold incomplete tasks in the task list is negligible next to the overall size of the dynamic programming tables, and shrinks as the size of the tables near their maximum, so rarely factors into maximum memory usage.

The extra memory added by assistant tables in ribosomal RNA leaves the maximum memory usage between 1.2 and 1.8 times the amount needed in the sequential implementation. The M1 RNA on the other hand, which features significantly more entries in the 8-dimensional table than the 4-dimensional one (leading to more assistant table data), used between 1.5 and 2.5 times the amount in the sequential implementation. These values do not change based on the number of processors used, minimizing the impact as more processors are added, meaning the implementation is very scalable with respect to memory usage.

4.2. Speedup and Efficiency

The average amount of speedup achieved by the parallel implementation is shown in Fig. 5. Speedup achieved by ribosomal and M1 RNA are on average very similar, with M1 RNA showing slightly more speedup at each point. Speedup is fairly modest overall, showing average speedup of only 6.5 to 7 on 64 processors. Average efficiency drops considerably as the number of processors increases as well, going from 0.37 on 4 processors to 0.11 on 64 processors.

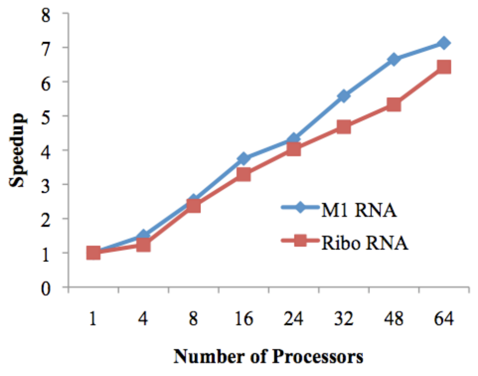

Figure 5. Average speedup achieved with 4-64 processors

The amount of speedup achieved becomes more variable as the number of processors increases as well, varying from 3.4 to almost 11 on 64 processors. This variation is due to the drastic variability of communication and load balancing time based on the properties of the input structures.

4.3. Understanding the Speedup and Efficiency Results

There is no all-encompassing cause of the modest levels of speedup and efficiency, but rather a series of causes stemming from the high memory usage, selective allocation and computation, and the non-serial polyadic nature and irregular dependency of the algorithm itself which make obtaining high levels of speedup impossible. Table 1 shows

Table 1. Percentage of Wall Clock Time by Activity

p	Calculation	Comm.	Polling	Load Balancing	Idling
1	100	0	0	0	0
4	45	2	40	8	5
8	31	2	44	16	7
16	24	3	45	19	9
24	19	3	48	20	10
32	17	2	48	21	12
48	13	2	48	22	15
64	11	2	49	22	16

the average amount of total wall-clock time performing each activity.

The first major cause for the modest levels of speedup is the inherent sequentiality imposed on the problem through selective allocation and computation, specifically the staggered start time for computation which cannot be avoided. The staggered start time, along with dynamic task creation, ensures that there are times when worker processors are *idle*, meaning their task list is empty and they are simply waiting for requests for work.

The second major cause of the lower levels of speedup is the necessity of load balancing. As the problem size and/or complexity increases, and as the number of processors increases, load balancing overhead becomes more of an issue. The two major causes of the lack of scalability of load balancing are the need to synchronize workers, and the increasing amount of data that can potentially be transferred. As total memory usage increases into the hundreds of gigabytes, it is not impossible to find a load balancing situation in which gigabytes of tabular data must be transferred.

The final major cause for the lowering of speedup is the additional task communication overhead brought on by having a non-serial polyadic dynamic programming algorithm with irregular dependency as the basis for the parallel design. While direct task communication is extremely quick due to the use of non-blocking communication to send short messages, the unpredictable nature of communication forces a side-effect on workers which is very time consuming.

This side-effect, known as "polling", occurs when an item in the main list which requested a result from another worker is checked to see whether the result has returned. If the result has not returned and therefore is not ready to be solved, the item is appended to the back of the list. This process is not time consuming on its own, but can begin to take large chunks of time as the number of list items waiting on communication increases.

4.4. Load Balancing

Load balancing decisions can be broken down into two categories for evaluation: those which are made to allow the program to continue execution (mandatory), and those which are made to increase the speedup and effciency of execution (discretionary). The former occurs when a processor is nearing the point at which it will no longer be able to continue evaluating tasks due to a lack of memory, and is a non-negotiable decision.

Load balancing decisions of the latter category are more interesting to study. As noted in section 3.4, discretionary load balancing decisions are designed to occur significantly more often during the beginning stages of execution when there are a large number of idling worker processors. Fig. 6 shows a comparison of the average amount of time spent idling for two different managers: one which performs discretionary load balancing during the early stages of the program, and one which only performs load balancing in non-negotiable situations.

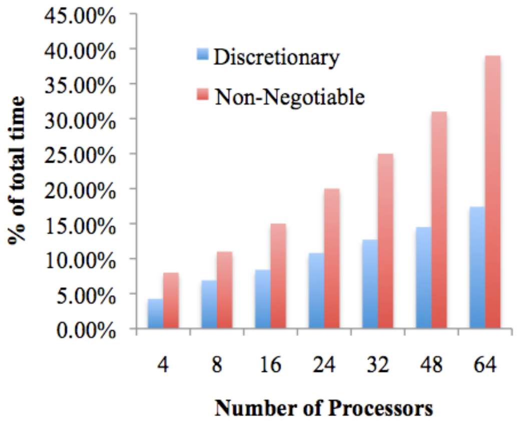

Figure 6. A graph of the difference in idle time as a percentage of total execution time based on a manager that is aggressive about load balancing during the early stages of execution versus one which is not.

Clearly, discretionary load balancing during early stages of execution is extremely beneficial, which is not surprising given that it is unlikely in a situation with a large number of processors that they will all become involved during the early stages without manager support.

Fig. 7 illustrates an example in which one manager performs discretionary load balancing throughout the execution of the program, while the other stops performing discretionary load balancing after the initial phase, and only performs non-negotiable load balancing thereafter. This shows that the efforts of the manager to add efficiency are overpowered by the overhead associated with load balancing after a certain point.

5. Conclusions

In this paper we have described the parallelization of a dynamic programming algorithm used to find the maximum common ordered substructure of a pair of RNA strands which include pseudoknots. The parallelization has to overcome several obstacles including the non-serial, polyadic

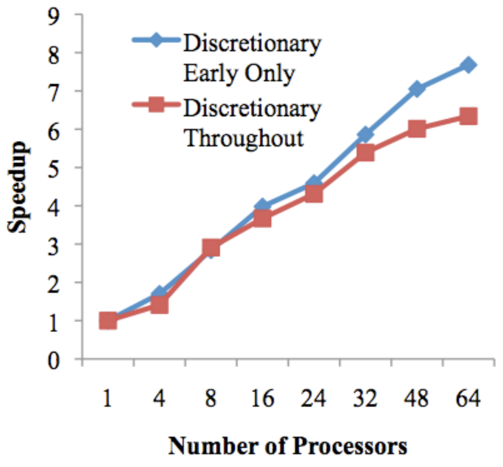

Figure 7. A graph of the difference in speedup based on a manager that performs discretionary load balancing during all stages of execution versus one which only does so during the early stages

nature of the algorithm and its irregular dependencies, and the selective calculation and allocation employed by the sequential implementation, which removes much of the inherent parallelism from the problem.

The parallelization uses the newly proposed *manager-worker* paradigm, which focuses on a hands-off manager acting as a monitor for load balancing should the need arise. Worker processors dynamically generate tasks and their own section of the distributed data structures while requesting results from other workers when necessary. Inter-worker task communication is designed to be flexible, allowing workers to deal with the unpredictable communication pattern while attempting to minimize parallel overhead.

The experimental results show a modest level of speedup with a highly-scalable level of memory usage, allowing larger and more complex input structures to be solved than were previously possible. While the results thus far do not give nearly the speedup that more easily-parallelizable problems are able to achieve, they do provide a necessary avenue of exploration for future work in parallelizing this type of problem.

References

[1] ACEnet: Atlantic Computational Excellence Network: http://www.ace-net.ca

[2] V. Bafna, S. Muthukrishnan, R. Ravi. Computing Similarity between RNA Strings. *DIMACS Technical Report*. Vol. 96, no. 30. 1996.

[3] J. Brown. The Ribonuclease P Database, *N*ucleic Acids Research. Vol. 27, 1999, p. 314.

[4] J. Cannone et al. The comparative RNA web (CRW) site: an online database of comparative sequence and structure information for ribosomal, intron, and other RNAs, *B*MC Bioinformatics. Vol. 3, no. 1, 2002, p. 15.

[5] C. Chen, B. Schmidt. An adaptive grid implementation of DNA sequence alignment, *Future Generation Computer Systems*. Vol. 21, no. 7, 2005, pp. 988-1003

[6] P. Evans. Finding Common RNA Pseudoknot Structures in Polynomial Time. *Combinatorial Pattern Matching, Lecture Notes in Computer Science*. Vol. 4009, 2006. pp. 223-232.

[7] A. Grama, A. Gupta et al. *Introduction to Parallel Computing, Second Edition*. Addison-Wesley, 2003.

[8] N. Jones, P. Pevzner. *An Introduction to Bioinformatics Algorithms*. MIT Press, 2004.

[9] W. Liu, B. Schmidt. Parallel Design Pattern for Computational Biology and Scientific Computing Applications, *IEEE International Conference on Cluster Computing, Proceedings*. 2003, pp. 456-459.

[10] W. Martins et al. Whole Genome Alignment using a Multithreaded Parallel Implementation. *13th Symposium on Computer Architecture and High Performance Computing, Proceedings*. September 2001.

[11] G. Parmentier, D. Trystram, J. Zola. Large scale multiple sequence alignment with simultaneous phylogeny inference. *Journal of Parallel and Distributed Computing*. Vol. 66, no. 12, December 2006, pp. 1534-1545.

[12] K. Zhang. Computing similarity between RNA secondary structures. *Proceedings of IEEE International Joint Symposia on Intelligence and Systems*. 1998, pp. 126-132.