# A Parallel, Out-of-Core Algorithm for RNA Secondary Structure Prediction

Wenduo Zhou
Department of Physics and Astronomy
University of Georgia
Athens, GA 30605, USA
wd_zhou@physast.uga.edu

David K. Lowenthal
Department of Computer Science
University of Georgia
Athens, GA 30605, USA
dkl@cs.uga.edu

## Abstract

*RNA pseudoknot prediction is an algorithm for RNA sequence search and alignment. An important building block towards pseudoknot prediction is RNA secondary structure prediction. The difficulty of extending the secondary structure prediction algorithm to a parallel program is (1) it has complicated data dependences, and (2) it has a large data set that typically cannot fit completely in main memory.*

*In this paper, we propose a new out-of-core, distributed-memory algorithm for RNA secondary structure prediction. Its novelty lies in its redundant file scheme, I/O-reducing in-core buffer mechanism, and dynamic load balancing algorithm. Experimental results obtained on 16 Sun UltraSPARC IIIi nodes provide evidence that our approach achieves good speedup. Furthermore, we found that counterintuitively, the* size *of the in-memory buffer is critical to efficiency of the parallel program.*

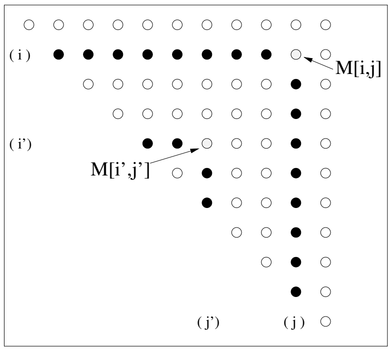


**Figure 1. Data dependence pattern for RNA Secondary Structure Prediction Algorithm. Matrix elements depend on all elements in the same row and column.**

## 1 Introduction

Due to the increasing number of new biological molecular sequences, it is becoming more attractive to determine the structure of them computationally. The RNA pseudoknot has been treated as a significant structural motif in a wide range of biological processes of RNAs. Stochastic context-free grammars (SCFGs) have been adapted from computational linguistics to develop an application to model RNA pseudoknotted structures with great success [3, 7]. An advanced grammar modeling approach, which is based on parallel communication grammar systems (PCGS), was introduced to specify pseudoknotted structures; it avoids context-sensitive rules and using a single CFG synchronized with a number of regular grammars [4]. This approach does not only permit automatic generation of a single-RNA structure prediction algorithm for each specified pseudoknotted structure model, but also makes it possible to develop full probabilistic models of pseudoknotted structures to allow the prediction of consensus structures by comparative analysis and structure homology recognition in database searches.

An important building block for RNA pseudoknot prediction is RNA secondary structure prediction, which is also computationally intensive. As the scale of the secondary structure prediction problem grows, its parallelization becomes an important issue. Furthermore, this algorithm has a potentially huge data set, meaning that the primary data structure must reside on disk; in other words, the program is likely to be *out of core* [2].

This paper focuses on developing a distributed-memory, out-of-core parallel program for RNA secondary structure prediction. The difficulties are that the data dependence makes it difficult to (1) determine an efficient data partitioning (distribution), and (2) determine an efficient computational pattern that limits the amount of I/O. Specifically, the data dependence pattern (shown in Figure 1) is such that the value of $M[i,j]$ depends on all matrix elements in row $i$ to the left, and all elements in column $j$ below. This is a dependence pattern that is atypical of most parallel programs that we are aware of and causes so-called “strided” data accesses [8, 15]—and, for this application, tail-end load imbalance when a row- or column-based distribution is used. Strided accesses are detrimental in in-core applications because of poor cache performance, but they are especially detrimental in out-of-core applications such as secondary structure prediction [8], because disk accesses are in general several orders of magnitude more expensive than memory accesses.

To counter these difficulties, our implementation has many novel attributes. First, we use a unique two-file approach to trade a small amount of extra computation for better locality. Second, we use an in-core buffer scheme that actually *reduces* the total amount of I/O. Finally, we use a dynamic load balancing scheme that improves performance by up to a factor of two.

Our results show two primary aspects of the parallel, out-of-core secondary structure prediction program. First, unsurprisingly, while speedup on small sizes is relatively poor, the speedup on large sizes is good. On a problem size as $8192 \times 8192$, we got a speedup of $6.1$ when using 8 nodes.

Second, we were surprised to find that the choice of the size of the in-core buffer (often called ICLA [2]) has a dramatic effect on performance—but not in the way one would generally expect. Specifically, beyond a point, increasing the ICLA size—which is generally thought to result in *lower* I/O overhead but the same computational cost—instead results in slower execution time. For example, the execution time for a problem of size $8192 \times 8192$ is $871$ seconds with a $2.5$ MB in-core buffer. As the size of in-core buffer was doubled to 5M, the execution time *increases* to $964$ seconds. This is because of the interaction of the data dependence pattern and cache hit rate—and is despite the other surprising attribute that increasing ICLA size actually *reduces* the *amount* of I/O. Overall, the tradeoff between choosing a small and large ICLA are reminiscent of choosing a tile size in pipelined parallel programs [11, 9, 12].

The rest of this paper is organized as follows. Section 2 describes the RNA secondary structure prediction algorithm and design issues in out-of-core parallel parallel programs. It also covers related work in both areas. Implementation details are given in Section 3, and the results of our experiments are presented and analyzed in Section 4. Finally, Section 5 summarizes this paper.

## 2 Background

While the RNA secondary structure prediction algorithm can be applied up to a 3-dimensional instance, our present work is to develop a 2-dimensional out-of-core parallel algorithm. The algorithm will compute all the elements along the same diagonal in parallel. The basic update formula [4] has the format:

$$M[i,j] = \begin{cases} f_1(i) & \text{if } i = j \\ \max_{k=i}^{j-1}(f_2(M[i,k], M[k+1,j])) & \text{if } i \neq j \end{cases}$$

This section gives background: it first describes secondary structure prediction in more detail and then discusses out-of-core, distributed-memory parallel programming.

### 2.1 RNA Secondary Structure Prediction Algorithm

In general, secondary structure in biochemistry and structural biology describes three-dimensional form of local regions. RNA secondary structure prediction is an important building block for RNA pseudoknots, which themselves are complicated sequences that are folded by interactions that determine the RNA molecule structure between pairs of nucleotides in the molecule. A system that is able to automatically generate a secondary structure prediction algorithm for each specified pseudoknotted structure model is created by using SCFG models. Similar to the well-known CYK [1] algorithm, the automated secondary structure prediction algorithm is based on dynamic programming [4]. As a sequence $x[1..n]$ is input, the maximum probability for every subsequence $x[i...j]$ is computed to permit the substructure specified by every nonterminal $X$. For the secondary structure, the maximum probability, in two dimensions, for $X$ to derive a pseudoknotted structure is [4]:

$$P(X,i,j) = \max_{Y,Z,i \leq k < j}\{P(Y,i,k) \cdot P(Z,k+1,j) \cdot P(X \rightarrow YZ)\}$$

This algorithm has a time complexity as $O(N^3)$, where $N$ is the length of sequence, as any matrix element $Pr(X,i,j)$ will be used $(N-i-1)+j$ times in order to compute the rest of the probability matrix.

### 2.2 Out-of-Core Data Parallel Programs

For achieving solutions with higher precision, problem sizes of scientific applications are typically increased. This may cause the primary data set to exceed main memory during computation. The typical solution is to write the code so that the data structure is stored on disk. A (usually) small part of each data structure will reside in memory at any given time. This is called *out-of-core* programming.

There are three approaches to out-of-core programming. The first one is to use virtual memory, which automatically ensures correctness for programs whose data size exceeds the size of available memory. But due to a lack of application-specific knowledge about the program's data dependence and parallelism, the operating system will usually page data suboptimally, causing poor performance [13]. There has been work done on making the virtual memory system more efficient for out-of-core programs by allowing user-designed memory managers [5]. Another solution is to allow programmers to write the code with explicit file I/O. This is the approach that we use. The last is compiler-directed explicit I/O [2, 10], which is designed to convert an in-core program to an out-of-core one via a compiler. However, this approach is not effective for programs with complicated data dependences. The data dependences of the secondary structure algorithm is along both rows and columns, while the computation is along diagonals—this is much more complicated than modern compilers or virtual memory systems can handle efficiently. Therefore, to develop a parallel out-of-core program, we will write the parallel out-of-core program with explicit file I/O.

The out-of-core parallel programming model we use is derived from the data-parallel programming paradigm [2]. A decomposition of the data domain exploits the inherent parallelism. The *Local Placement Model* [2] is applied. In this model, each processor has a disk that acts as another level of memory, i.e., extending the well-known SPMD model [6].

In the local placement model, the data structure of each processor resides on disk in a logically separate file called *Local Array File* (LAF) or *Out-of-Core Local Array* (OCLA) [2]. The portion of the data structure loaded into memory for computation is called the *In-Core Local Array* (ICLA). No processor is allowed to access directly to any other OCLA. The only approach for a processor to obtain the data in another processor's OCLA is via communication after the data is fetched.

## 3 Implementation

Given the formula for $M[i, j]$ defined previously, the computation will start from the main diagonal (i.e., $M[i, i]$ for all $i$ , assuming that M is an $N \times N$ matrix) and proceed to the upper right corner of the matrix (i.e, $M[1, N]$). One legal execution order is to compute all matrix elements on one diagonal in each iteration.

There is no data dependence between any pair of matrix elements on the same diagonal. Therefore, all the matrix elements on same diagonal can be computed in parallel. To compute element $M[i, j]$, all the (previously computed) points in row $i$ to the left and in column $j$ below are required. Note that dynamic programming is not possible, as no partial results can be stored from previous iterations.

In many parallel algorithms, the parallelism is solely along columns (or rows), while the data dependence is along rows (or columns). Thus, the scheme to partition the matrix is straightforward and minimizes the communication. However, the problem that we are facing has parallelism along the diagonal of the matrix, but has the data dependences along both rows and columns. While we initially use a standard row-wise data distribution, the data dependences make the implementation challenging.

In this section, we first solve the in-core distributed-memory parallel algorithm to study the communication pattern. Then we propose an out-of-core sequential algorithm to choose an effective disk file layout and access pattern. A "two-file" method is found the most appropriate for the RNA secondary structure prediction algorithm. Then, an out-of-core parallel algorithm is developed combining both previous algorithms.

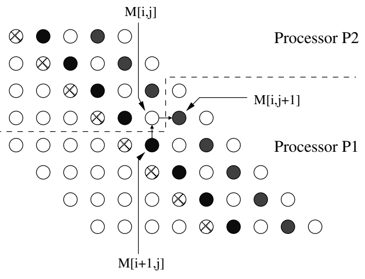


**Figure 2. Row and column dependencies.**

### 3.1 In-Core Distributed-Memory Parallelization

We will first discuss the simpler problem of parallelizing secondary structure prediction a distributed-memory program, but is *in core*. (We will not discuss the shared memory version, as it has previously been developed [14].)

As stated above, communication is the critical issue for the in-core parallel algorithm. The communication patterns along row and column are similar. Communication may be initiated as soon as any matrix element on the boundary is computed. If this element is on the right boundary, the processor will send all the elements that have been computed in that row to its right neighbor. On the other hand, if the element resides on the upper boundary, its processor should send all the matrix along that column to its upper neighbor. For example, as shown in Figure 2, $M[i, j]$ is on the right and lower boundary of the sub matrix assigned to processor $P_2$. Both $M[i, j+1]$ and $M[i+1, j]$ belong to processor $P_1$. $M[i, j]$ is computed one iteration earlier than $M[i, j + 1]$, which is the first element in row $i$ to be computed by $P_1$. Therefore, as soon as $M[i, j]$ is computed, all elements in row $i$ that are to the right of $M[i, j]$ should be sent to $P_1$ in order to allow $P_1$ to compute $M[i, j + 1]$ in the next iteration. The similar situation also occurs along the column, such that $M[i + 1, j]$ is computed one iteration earlier than $M[i, j]$. Thus, $P_1$ must transmit to $P_2$ all elements on column $j$ as soon as they have been computed.

The amount of communication for rows and columns is identical because their communication patterns are the same. Thus, we only need to estimate the amount of communication along column or row. Here, we calculate the amount of communication along the columns. Assume $P_0$ is the bottom processor (it sends data to its upper neighbor $P_1$). Meanwhile, $P_1$ sends its elements, as well as those from $P_0$, to $P_2$. This continues until the processor that is second from the top sends to the top processor all the elements that the top processor does not own. During execution, $P_0$ sends all elements allocated to it to $P_1$, which sends both the elements of it and those elements of $P_1$ to $P_2$, and so on. Generally, one processor will send all the matrix elements assigned to itself and the processors below it to its upper neighbor. The amount of communication, $C$,

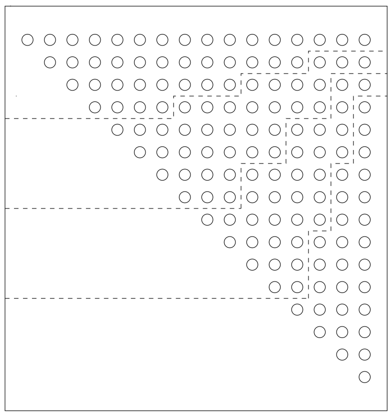


**Figure 3. Data distribution after dynamic load balancing.**

is $O(N^2 \cdot P)$. (For full derivational details, see [16].) Thus, the communication along columns or rows is $O(N^2 \cdot P)$ versus the total computation time of $O(N^3)$. As long as $N \gg P$, good speedup should be obtained.

**Dynamic Load Balancing** Unfortunately, a row- or column-wise data distribution causes tail end load imbalance. Hence, we employ the following scheme for dynamic load balancing: the "lower" processors, once they complete their work, share the load of the "higher" processors. The dynamic load balancing occurs between iterations.

Specifically, we use a straightforward load balancing technique, which works as follows: if the total number of processors is $P$, processor $i$ ($i > 0$) sends $(P - i) \times \alpha$ rows to processor $i - 1$, where $\alpha$ is 1 (for the in-core version). This is repeated every $P \times \alpha$ diagonals. This simple scheme performed well in practice for the in-core version.

The overhead of load balancing is $O(N^2)$, which is small compared to the $O(N^3)$ overall computation time. An example partitioning after load balancing is shown pictorially for four processors in Figure 3. This load balancing is necessary—our results show that for large matrix sizes, the execution time with dynamic load balancing is a factor of two faster than the corresponding time without it.

## 3.2 Out-of-Core Sequential Algorithm

As our second step towards our desired distributed-memory, out-of-core parallel program, we investigate an out-of-core algorithm that is *sequential*. We discuss the in-core buffer structure, the disk structure, and how I/O is done.

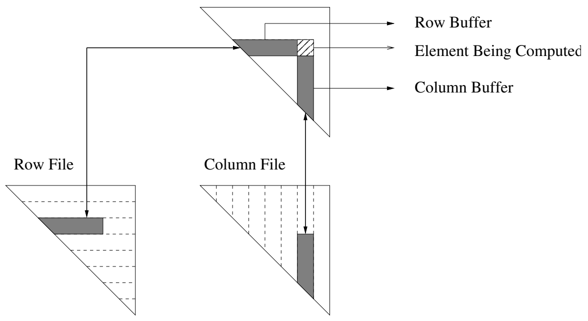


**Figure 4. Mapping between in-core buffer and disk file. The shaded square area represents both row buffer and column buffer, which are updated together.**

### 3.2.1 Data Structure for In-Core Buffer

The data dependence is along both row and column (shown in Figure 4). Therefore, for efficient access, we keep parts of rows and columns in memory. These buffers hold the data read from disk for computing one or more matrix elements inside both those rows and columns. The in-core buffer (also called ICLA [2]) is composed of row buffer and column buffer. We denote the number of rows and columns in these buffers as $k_r$ and $k_c$, respectively.

### 3.2.2 File Structure of Out-of-Core Buffer

Because the matrix has data dependences along both rows and columns, a single file containing the whole matrix is not a good solution. For example, if a single file is used to store the matrix in row major, when a column is required the whole file will need to be read to collect the scattered data. The identical problem also occurs if we store the data in column major.

Therefore, we use two files to store the matrix. Both files contain the exactly same matrix. One stores the matrix in row major, while the other one is in column major. Therefore, the situation to read through the whole file for small amount of scattered data is avoided. Both files are be updated after a subset of matrix elements in the matrix are computed (also shown in Figure 4).

### 3.2.3 I/O Scheme

We must choose an ICLA scheme to minimize I/O, which is generally consumes a significant amount of time in an out-of-core program. Therefore, we strive to choose an ICLA scheme with the least I/O cost.

Denote $R$ as the number of file read operations. The in-core buffer is composed of $k$ arrays, each of which contains $N$ elements, where $N$ is the matrix size. Among those $k$ arrays, $k_r$ arrays are allocated to rows, while rest ($k_c$) are

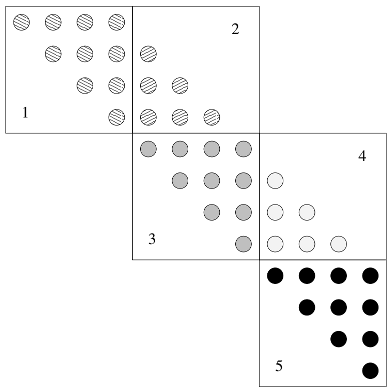


**Figure 5. Pictorial view of our I/O scheme. Circled elements in each window (square) are computed using the same buffer.**

allocated to columns. In general, we have $k_r \cdot N + k_c \cdot N \leq S_M$. where, $S_M$ is the size of total available memory of the local processor.

We developed 3 ICLA schemes: minimum-replacement, complete-replacement and half-replacement. They are discussed in full in an accompanying technical report [16]. Due to space limitations, we discuss only half-replacement in this paper—it was the best of the three algorithms.

With *half replacement*, we are able to read or write several rows or columns in one operation and compute several diagonals in a single iteration. In each I/O operation, $k_c$ columns are read into the column buffer, flushing the previous contents. Meanwhile, only $\lfloor \frac{k_r}{2} \rfloor$ rows are read into the row buffer, which replace the oldest $\lfloor \frac{k_r}{2} \rfloor$ rows. With this scheme, $\lfloor \frac{k_r}{2} \rfloor$ diagonals can be computed in one iteration.

For example, in Figure 5, $k_r = 8$ and $k_c = 4$. Assume that rows $i-4, i-3, i-2$ and $i-1$ are already in the row buffer. First, rows $i, i+1, i+2$ and $i+3$ are read. After matrix elements in window 1 are computed, rows $i+4, i+5, i+6$ and $i+7$ are read in, replacing row $i-4, i-3, i-2$ and $i-1$. When all elements in those four diagonals on row $i, i+1, i+2, i+3$ are computed, they are written back to column file, including the matrix elements in window 2. On the other hand, there is no need to write them back to the row file, because no elements in window 4 have been computed. After all the elements in window 4 are computed, elements in window 3 and 4 will be written back to the row file. The elements in window 4 depend on elements in window 5. This requires that $k_r$ rows be in memory together.

Analyzing this scheme, we see that for the first $\lfloor \frac{k_r}{2} \rfloor$ diagonals, there are $\lceil \frac{N}{\lfloor k_r/2 \rfloor} \rceil$ reads for rows and another $\lceil \frac{N}{k_c} \rceil$ reads for columns. For the second diagonals, there should be $\lceil \frac{N-\lfloor k_r/2 \rfloor}{\lfloor k_r/2 \rfloor} \rceil$ reads for rows and $\lceil \frac{N-\lfloor k_r/2 \rfloor}{k_c} \rceil$ reads for columns. This continues until the last few ($\leq k_r$) diagonals, which has one read for rows and $\lceil \frac{k_r}{k_c} \rceil$ reads for columns. Hence, we get the following.

$$R \quad = \quad \sum_{i=2}^{\frac{N}{\lfloor k_r/2 \rfloor}} \left( \lceil \frac{N - i \cdot \lfloor k_r/2 \rfloor}{\lfloor k_r/2 \rfloor} \rceil + \lceil \frac{N - i \cdot \lfloor k_r/2 \rfloor}{k_c} \rceil \right)$$

which is $O(N^2 \cdot (\frac{1}{k_r^2/2} + \frac{1}{k_c \cdot k_r}))$.

Because $W = R$ ($W$ is the number of file writes), the number of total I/O events is $O(N^2 \cdot (\frac{1}{k_r^2/2} + \frac{1}{k_c \cdot k_r}))$. Furthermore, as above, the total amount of I/O is:

$$D_r \quad = 2 \times \sum_{i=2}^{\lceil \frac{N}{\lfloor k_r/2 \rfloor} \rceil} [N - (i-1) \cdot \lfloor \frac{k_r}{2} \rfloor] \cdot (i-1) \cdot \lfloor \frac{k_r}{2} \rfloor$$

which is $O(\frac{N^3}{k_r})$. The amount of data that is written is neglected, as its time complexity is only $O(N^2)$.

It is important to note that according to the above analysis, the total amount of data read, $D_r$, *decreases* by $1/k_r$ as $k_r$ increases. This is unlike any out-of-core parallel program that we have seen.

Assuming that the total in-core buffer size is fixed, there is still a degree of freedom to choose the size of the row and column buffers. Suppose that the in-core buffer size contains a total of $k$ rows and columns. One possible allocation is to use one a one-column buffer and a $k-1$ row buffer. Then the amounts of data through I/O can be reduced to the minimum. However, increasing the row buffer size increases the number of I/O operations. For example, if we choose $k_r = 2k/3$ and $k_c = k/3$ (as we did in our experiments), then the number of I/O operations is reduced. Another concern is that a larger row buffer size may cause more load imbalance, because the row buffer size determines when our data redistribution scheme (which balances the load during program execution) terminates. However, given our data redistribution scheme, this effect is not significant because we use a relatively small in-core buffer size. This is discussed further below and in the next section.

The pseudo-code for our scheme is given below. For clarity, we call the first half of the row buffer $rowbuf_1$ and the second half $rowbuf_2$. Also, the column buffer is called $colbuf$.

The total number of iterations, $I$, is $\lceil \frac{N}{\lfloor k_r/2 \rfloor} \rceil$. On iteration $i$, the longest diagonal to compute is diagonal $i \cdot \lfloor k_r/2 \rfloor$, which contains $N - i \times \lfloor k_r/2 \rfloor$ matrix elements. The algorithm follows.

```
1: for all $i = 0$ to $I - 1$ do
2:   Diagonal number $d = i \times \lfloor k_r/2 \rfloor$
3:   $n_{cc} = 0$ (# of uncomputed columns in $colbuf$)
4:   for all $g_r = 0$ to $N - i \cdot \lfloor k_r/2 \rfloor$ with step $\lfloor k_r/2 \rfloor$ do
5:     Read elements from row $g_r$ through row $g_r + (\lfloor k_r/2 \rfloor - 1)$ into $rowbuf_2$.
6:     if $n_{cc} \geq \lfloor k_r/2 \rfloor$ then
7:       $r_c = 1$
8:     else
```

```
      **if** $n_{cc} = 0$ **then**
        $r_c = \lceil \frac{\lfloor k_r/2 \rfloor}{k_c} \rceil$
      **else**
        $r_c = \lceil \frac{\lfloor k_r/2 \rfloor - n_{cc}}{k_c} + 1 \rceil$
      **end if**
    **end if**
    **for all** $g_c = 1$ to $r_c$ **do**
      **if** $n_{cc} = 0$ OR $g_c \neq 0$ **then**
        Read $\min(k_c, \lfloor k_r/2 \rfloor)$ cols into $colbuf$
      **end if**
      **for all** $r = 0$ to $\min(k_c - 1, \lfloor k_r/2 \rfloor - 1)$ **do**
        Compute $\lfloor k_r/2 \rfloor$ elements in next column
        above diagonal $d$ in $colbuf$
      **end for**
      Update $n_{cc}$
      **if** $n_{cc} = 0$ **then**
        Write elements in $colbuf$ to column file
      **end if**
    **end for**
    Write computed elems in $rowbuf_1$ to row file
    Switch $rowbuf_2$ and $rowbuf_1$
  **end for**
  Write computed elements in $rowbuf_1$ to row file
**end for**
```

## 3.3 Distributed Memory, Out-of-Core Algorithm

The out-of-core parallel secondary structure prediction algorithm is a (relatively) straightforward combination of both the in-core parallel algorithm and out-of-core sequential algorithm. The algorithm in its entirety is given in our accompanying technical report [16]. Here, we focus only on the differences from the algorithm for the out-of-core, sequential algorithm. They are as follows.

- Each processor iterates over its section (set of rows) of the matrix (as is typical in many parallel programs).
- An in-core buffer must be allocated to hold communicated (received) data; if it cannot be held in memory, it must be stored on disk.
- Load balancing must be done. The idea is similar to the distributed-memory, in-core version. The difference is that the value of $\alpha$ is $k_r/2$ instead of 1 (as was used in the in-core version). See [16] for full details.

# 4 Performance

This chapter discusses the results of our experiments. All results were obtained using a cluster of 16 Sun workstation nodes. Each node was a 1 GHz UltraSPARC IIIi processor, with a 64K/32K (data/instruction) split L1 cache, 1MB L2 cache and 512 MB main memory. Tests were run when the only other system activities were daemon processes.

| $N$ | One-File Time (s) | Two-File Time (s) |
|---|---|---|
| 100 | 1.77 | 0.12 |
| 200 | 14.05 | 0.58 |
| 400 | 112.77 | 2.28 |

**Table 1. One- versus two-file approach.**

If the RNA secondary structure prediction program is in-core, the primary data structure is the upper triangular part an $N \times N$ matrix of doubles, which means that the total memory consumed is $S = 4 \times N^2$ bytes, since the elements are double-precision numbers. When the program is out of core, the row buffer contains $k_r$ rows ($N$ elements per row), and the column buffer contains $k_c$ columns; typically, $k_r \ll N$. Hence, the total size of the in-core buffer (the ICLA) in the sequential program is $8 \cdot (k_r + k_c) \cdot N$. For the parallel program, another buffer containing $k_r$ rows is required to hold values communicated from neighboring processors; hence, the in-core buffer size in bytes for the parallel program is

$$S_{ic} = 8 \cdot (2k_{row} + k_{col}) \cdot N.$$

## 4.1 Benefit of Two-File Approach

Section 3.2.2 explains that we use a two-file approach, along with redundant computation, to avoid strided file access. (We cannot measure the time for the entire program because it takes so long that there is interference from other processes.) Table 1 shows results from a program that simulates the access patterns of the secondary structure prediction code. The table shows that the execution time is two orders of magnitude smaller with the two-file approach.

Note that the cost for faster disk access is twofold: first, there is extra space consumed on the disk due to the extra file, which we believe is a good trade-off given the size of modern storage systems. Second, there is an extra update operation for each point. The number of these extra operations is linear in the total number of matrix elements, in that each matrix element is exactly updated twice instead of once. Still, the two-file approach is vastly superior; this is because compared with the $O(N^3)$ prediction algorithm, the extra update operations are negligible.

## 4.2 Speedup Results

We measured speedup over problem sizes of $N = 1K$ through $N = 16K$, covering each power of 2. At 8K and 16K, the problem is out of core. The ICLA size was approximately $\frac{48}{N}$ of the total problem size, which corresponds to roughly 5% to less than 1%, depending on the problem size. When the problem size doubles, the execution time is increased about 8 fold. The raw times verify that RNA secondary structure prediction is a cubic-time algorithm.

We show a graph of speedup in Figure 6. When the prob-

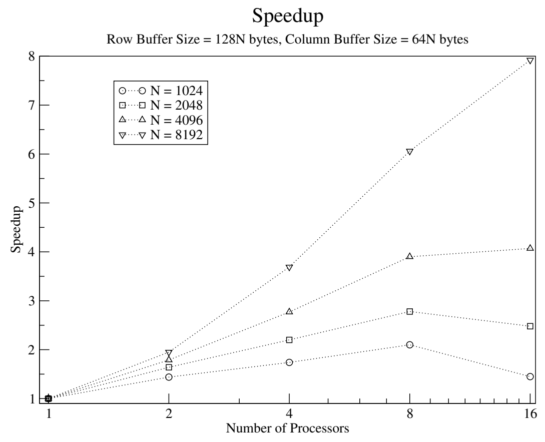
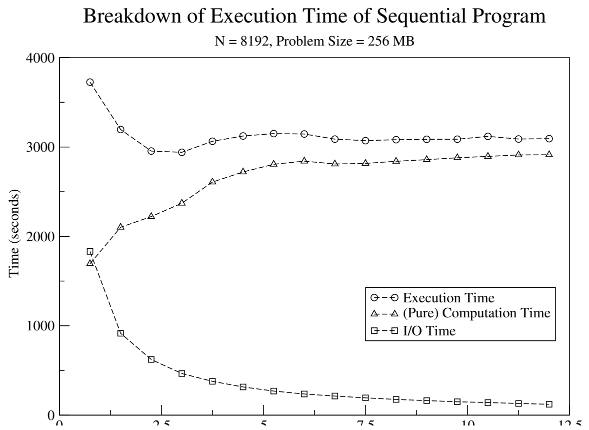


**Figure 6. Speedup with different sizes.**

| *Total ICLA Size (MB)* | *Time (seconds)* | | |
|---|---|---|---|
| | Computation | I/O | Total |
| 0.4 (12) | 212.7 | 181.3 | 408.8 |
| 1 (36) | 227.9 | 61.67 | 313.8 |
| 2 (60) | 249.1 | 38.01 | 305.2 |
| 3 (108) | 289.8 | 22.66 | 328.4 |
| 4 (132) | 318.4 | 18.36 | 351.9 |
| 5 (156) | 335.0 | 16.06 | 365.5 |
| 6 (180) | 347.0 | 14.01 | 374.6 |

**Table 2. Breakdown of times for $N = 4096$. The total time includes startup time that is not included in either computation or I/O.**



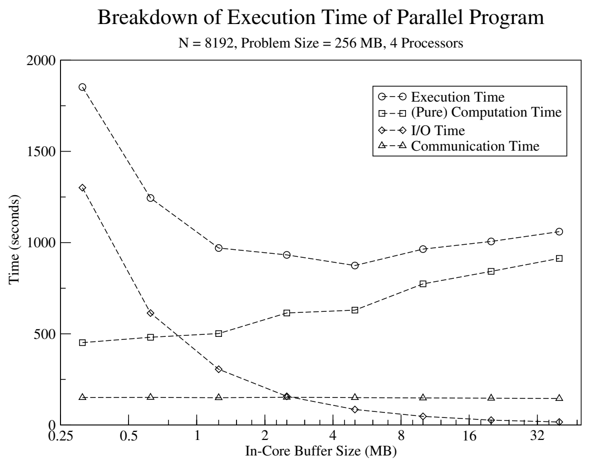


**Figure 7. Effect of in-core buffer size ($S_{ic}$). The top graph is of the sequential program, and the bottom one is a parallel program.**

lem size is small, for example, 16 MB ($N = 1024$) or 64 MB ($N = 2048$), speedup is poor because overhead (communication) dominates computation. On the other hand, when $N$ is large ($N \geq 4096$), speedup is nearly linear in the number of processors, as computation dominates.

The in-core buffer size, denoted $S_{ic}$, has a significant impact on the speed of the out-of-core program. We performed detailed tests for $N = 4096$ and $N = 8192$, with various in-core buffer sizes. Both sequential and parallel programs were executed to measure the (pure) computation time, I/O time, and communication time (if applicable). Figure 7 shows the result of the sequential out-of-core program with matrix size $N = 8192$ (top), as well as the result for a parallel out-of-core program for $N = 8192$ (bottom), respectively. Table 2 shows the result of the sequential out-of-core program with matrix size $N = 4096$.

The tests show that when the in-core buffer size increases, the (pure) computation time grows gradually, and the I/O operation time decreases. This is a counterintuitive result in general, and is in contrast to the performance of typical out-of-core programs. The expectation is that as the ICLA increases, I/O time is monotonically non-increasing, and computation time is relatively constant.

The reason why the pure computation time increases with in-core buffer size is due to an increasing number of cache misses. This is in turn caused by the unique data access pattern of the secondary structure prediction algorithm.

Specifically, the data accesses pattern is a "zig-zag" one. A full explanation can be found in [16], which we omit due to space considerations. The basic idea is that a more localized data access pattern results from a *smaller* in-core buffer size. Essentially, the larger the in-core buffer, the more cache misses that result.

Another typical aspect of out-of-core programs is that as long as the ICLA is "large enough", the I/O overhead is negligible. In other words, the I/O cost is dependent primarily on the matrix size, rather than on the ICLA size. However, our experimental results (Figure 7) show that the I/O time drops significantly as the row buffer size increases.

This is because, as discussed in Section 3, $T_{I/O} \propto \frac{1}{S_{ic}}$, for a fixed problem size $S$. This phenomenon is caused by the algorithm for ICLA scheme, in which the *total* amount

| N = 8192 | | | N=4096 | | |
|---|---|---|---|---|---|
| $S_{ic}$ | Time | $\Delta T$ | $S_{ic}$ | Time | $\Delta T$ |
| 0.3 | 1853 | 0.030 | 0.3 | 176.0 | 0.022 |
| 0.6 | 1244 | 0.040 | 0.6 | 143.9 | 0.038 |
| 1 | 970.2 | 0.740 | 1 | 126.9 | 0.083 |
| 3 | 932.5 | 1.67 | 3 | 123.3 | 0.200 |
| 5 | 874.3 | 4.42 | 5 | 118.7 | 0.554 |
| 10 | 964.3 | 12.8 | 10 | 135.0 | 1.94 |
| 20 | 1006 | 44.7 | 20 | 152.1 | 7.41 |

**Table 3. Load imbalance with various in-core buffer sizes for four-processor experiments. Sizes are in MB, and times are in seconds.**

of data brought into memory via I/O operations is determined by the in-core buffer size. This is also in contrast to typical out-of-core programs.

The in-core buffer size also determines at which iteration we cease balancing the load via data redistribution. The larger the in-core buffer, the earlier we stop. In Table 3, $\Delta T$ is defined as the largest difference in execution time among all processors. The table shows that when the in-core buffer size is small, the load imbalance is negligible. For example, for $N = 8192$ and $P = 4$, when $S_{ic} = 3$ MB, the load imbalance is $\Delta T = 4.42$ seconds, while execution time is $874.26$ seconds. On the other hand, for large row buffer sizes, $\Delta T$ can be over 15% of total execution time. As we have already shown that best performance is achieved for modest in-core buffer sizes, load imbalance will not be a significant problem.

Therefore, the relationship between in-core buffer size and execution speed is not monotonic. On one hand, the in-core buffer size should be large because less I/O will be performed. On the hand, if the in-core buffer size exceeds the cache size, cache misses can be a significant cost. Our experience indicates that an effective in-core buffer size is close to the cache size.

## 5 Summary and Future Work

In this paper, we have presented an approach to extend the RNA secondary structure prediction algorithm to a distributed-memory, out-of-core parallel version. Our implementation uses several novel techniques, including (1) a unique two-file approach to trade a small amount of extra computation for better locality, (2) an in-core buffer scheme that actually *reduces* the total amount of I/O, (3) a unique matrix partitioning scheme, and (4) a data redistribution scheme to balance load.

Experimental results showed that if the problem size is small, such that it is not necessary to be out-of-core, we cannot obtain a good speedup. If the problem size is large enough, the program is able to achieve good speedup. For instance, the speedup is 6.1 at 8 processors for an $8192 \times 8192$ problem. We also found that that the in-core buffer size has a significant impact on the performance. Counterintuitively, the computation time *increases* and the I/O time *decreases* as the in-core buffer size increases.

One area for future work is to focus on allocation of the in-core buffer. In particular, given a fixed size buffer, there is an additional dimension that we have not fully explored—the choice of the in-core row buffer size versus the in-core column buffer size. This tradeoff could further affect execution time, and we intend to address this issue more fully.

## References

[1] CYK algorithm. `http://en.wikipedia.org/wiki/CYK_algorithm`.

[2] R. Bordawekar, A. Choudhary, C. Koelbel, and M. Paleczny. A model and compilation strategy for out-of-core data parallel programs. In *Principles and Practice of Parallel Programming*, 1995.

[3] M. Brown. Small subunit ribosomal RNA modeling using stochastic context-free grammars. In *Intelligent Systems for Molecular Biology*, 2000.

[4] L. Cai, R. L. Malmberg, and Y. Wu. Stochastic modeling of RNA pseudoknotted structures: A grammatical approach. In *Intelligent Systems for Molecular Biology*, 2003.

[5] O. Cozette, A. Guermouche, and G. Utard. Adaptive paging for a multifrontal solver. In *International Conference on Supercomputing*, 2004.

[6] S. Hiranandani, K. Kennedy, and C.-W. Tseng. Compiling Fortran D for MIMD distributed-memory machines. *Communications of the ACM*, 35(8):66–80, Aug. 1992.

[7] I. Holmes and D. Rubin. Pairwise RNA structure comparison with stochastic context-free grammars. In *Pacific Symposium on Biocomputing*, 2002.

[8] D. Kotz. Disk-directed I/O for MIMD multiprocessors. In *Operating Systems Design and Implementation*, 1994.

[9] D. K. Lowenthal. Accurately selecting block size at runtime in pipelined parallel programs. *International Journal of Parallel Programming*, 28(3):245–274, June 2000.

[10] M.Kandemir, J.Ramanujam, and A.Choudhary. Improving the performance of out-of-core computations. In *Proc. International Conference on Parallel Processing*, Aug. 1997.

[11] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Rice University, Jan. 1993.

[12] R. F. Van der Wijngaart, S. R. Sarukkai, and P. Mehra. The effect of interrupts on software pipeline execution on message-passing architectures. In *International Conference on Supercomputing*, May 1996.

[13] D. Womble, D. Greenberg, S. Wheat, and R. Riesen. Beyond core: Making parallel computer i/o practical. In *Dartmounth Inst. for Adv. Grad. Studies*, 1993.

[14] Y. Wu. Implementation improvements to an RNA pseudoknot prediction algorithm. Master's thesis, Department of Computer Science, University of Georgia, 2003.

[15] C. Zhang and S. A. McKee. Hardware-only stream prefetching and dynamic access ordering. In *International Conference on Supercomputing*, 2000.

[16] W. Zhou and D. K. Lowenthal. A parallel, out-of-core algorithm for RNA secondary structure prediction. Technical Report 04-06, University of Georgia, Oct. 2004.