Exploring New Applications for Parallel Programming

Envisioning a New Realm

David Tepper
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td>3</td>
</tr>
<tr>
<td>An Introduction to Parallel Programming</td>
<td>3</td>
</tr>
<tr>
<td>What is Parallel Programming?</td>
<td>3</td>
</tr>
<tr>
<td>Parallel Programming Nuances</td>
<td>3</td>
</tr>
<tr>
<td>The Goals of a Parallel Algorithm</td>
<td>4</td>
</tr>
<tr>
<td>ZPL</td>
<td>6</td>
</tr>
<tr>
<td>Practical Example</td>
<td>12</td>
</tr>
<tr>
<td>Application</td>
<td>13</td>
</tr>
<tr>
<td>Gene-DNA Sequencing</td>
<td>14</td>
</tr>
<tr>
<td>Prohibitive Details</td>
<td>15</td>
</tr>
<tr>
<td>Industry Solution</td>
<td>15</td>
</tr>
<tr>
<td>Solution</td>
<td>16</td>
</tr>
<tr>
<td>Steps Toward a Result</td>
<td>17</td>
</tr>
<tr>
<td>Performing the Search</td>
<td>17</td>
</tr>
<tr>
<td>Result Optimizations</td>
<td>21</td>
</tr>
<tr>
<td>Speed Optimizations</td>
<td>26</td>
</tr>
<tr>
<td>Output Optimizations</td>
<td>29</td>
</tr>
<tr>
<td>Results</td>
<td>31</td>
</tr>
<tr>
<td>Impact and Proposed Usage</td>
<td>36</td>
</tr>
<tr>
<td>Summary</td>
<td>36</td>
</tr>
<tr>
<td>Acknowledgements and Awards</td>
<td>36</td>
</tr>
<tr>
<td>References</td>
<td>36</td>
</tr>
</tbody>
</table>
Working with Professor Larry Snyder, the focus of my research has been to learn and develop techniques for designing and implementing parallel algorithms, that is, algorithms which are specifically engineered for use in multi-processor systems. The over-arching idea behind the research was to learn these techniques and apply them toward real world problems in hopes of improving the performance of often used algorithms as multi-processor computers become increasingly available and common. After months of learning such techniques, I decided to apply them toward the problem of locating a gene within a DNA strand, currently an important research topic in the parallel-programming world.

As the sole contributor to this project, I designed and implemented a Gene-DNA sequencing algorithm, with instruction from Professor Snyder on the techniques used to create efficient parallel programs. Unfortunately, speed-testing the algorithm against the industry standard proved prohibitively expensive; however, should this algorithm prove to be faster, the impact on the field of Computational Biology would be substantial. The Human Genome Project began sequencing haploid genes within humans in 1990, and completed this endeavor in 2003, thus spending 13 years searching human DNA for gene information. With human diploid genes yet to be mapped, a faster searching algorithm would vastly expedite the process.

AN INTRODUCTION TO PARALLEL PROGRAMMING

WHAT IS PARALLEL PROGRAMMING?

Simply put, a parallel program is one in which multiple processors are working in concert on the same code segment. Each processor maintains some portion of the result within its local memory and operations are executed on all processors simultaneously, but not necessarily in lock step. It is important to note that this is substantially different from multi-threaded programs, where multiple processors may be operating on different portions of the code, but not all operations are guaranteed to be executed by all threads.

Parallel programs are particularly useful for tackling large data-manipulation tasks, as the data to be operated on can be spread across a pool of processors, allowing each processor to work on a small portion of the problem. This has the effect of allowing each processor to take full advantage of its cache and main memory, limiting secondary storage I/O, and simultaneously increasing the available processing power.

PARALLEL PROGRAMMING NUANCES

The point of having multiple processors perform the same operations is that even though they are executing the same instructions, they are doing so on different data. Thus, not all operations will function in the same way across all processors.
For example, consider an “IF” statement executed on the two processors depicted in the following:

```c
if (Data == “1”) then
doSomething();
end;
```

This operation will execute differently on either processor, as only Processor 1’s data will validate the statement’s condition. It is thus feasible for multiple processors to execute different code paths based on their data. Given this possibility, unless the input data to the parallel program is well-known ahead of time, the parallel program author cannot make assumptions as to which processors (or how many) will enter a given code path.

As another example of this type of event, consider a “WHILE” statement executed on the two processors depicted in the following:

```c
while (Data < 100) then
  Data++;
end;
```

In this example, the two processors will proceed to the next section of the code at different times, because Processor 1 will be executing the while loop while Processor 2 moves on. Thus it becomes clear that when writing parallel programs, unless the input data is well-known ahead of time, the author can also make no assumptions about the concurrency of operations across processors, or about temporal locality between operations on different processors. However, unlike multi-threaded applications, constructs such as spin locks, mutexes, semaphores, and monitors are not used (and are not necessary) to enforce any kind of serialized access to the data, because, again, the data is purely local to the individual processor.

For the purposes of this research, this parallel programming concept is known as “Shattered Control Flow” (SCF) and governs many of the design decisions involved in crafting a parallel solution.

Unfortunately, due to the notion of SCF and other considerations discussed in the following, programs that operate sequentially (that is, on a single processor or on multiple processors via threads) typically cannot easily be made into a parallel equivalent because, aside from simply changing the code that comprises the program, for an application to be truly parallel, it must undergo a fundamental change in its algorithm.

### THE GOALS OF A PARALLEL ALGORITHM

To understand how the goals of a parallel algorithm differ from those of a sequential algorithm, it is important to understand what the goals of a sequential algorithm are. In their simplest and most obvious form, the two major goals of a sequential algorithm are:

- **Correctness** – The algorithm must accomplish the goal for which it was designed, providing proper results as required by the application.
- **Computational Efficiency** – The Computational Complexity of the algorithm (usually expressed in “Big-O” notation) should be as minimal as possible while still preserving the correctness of the algorithm.
However, the matter is complicated for parallel programs. The primary reason for the complication is that the Computational Complexity of the algorithm is no longer the only governing factor for its efficiency. There are two other considerations that the would-be parallel programmer must take into account in order to design a truly efficient parallel algorithm.

### SIMULTANEOUS WORKLOAD

This gauge of computational efficiency is the measure of the amount of load sharing between processors. Given a large task and a pool of processors, it does not make sense to have one processor perform all the work while the others sit idly by. It is similarly inefficient to have a processor waiting for another to finish before it can perform its own work. In the case of data-manipulation style problems, the data should be operated on simultaneously by the entire processor pool. If any operations are necessarily based on the results of previous operations, then these operations should also be delegated to the entire processing pool. An algorithm which seeks to be “parallel-efficient” must spread out the entire workload to its entire pool of processors as evenly as possible.

The method through which the tasks are split evenly amongst the processor pool is not always obvious; indeed, achieving a high-degree of parallelism is a task of its own, and is discussed in more detail in the “Problem Space Promotion” section.

### COMMUNICATION

In many cases it is necessary for processors to exchange information during runtime, and while this may be a very useful tool, it has a significant effect on the running time of a parallel application. The problem is that, while the processors themselves may be incredibly fast (and indeed are growing faster at an amazing rate), the bus speeds connecting them are not nearly as quick. Thus, Communication steps in the algorithm force the processor to perform an incredibly slow I/O operation, which usually halts any further progress until it is completed. An interesting and very important fact is that because Communication between processors is so slow compared to the processing speed of the chips themselves, and the notion that in a parallel environment there are many such processors all operating simultaneously on the workload (assuming the appropriate amount of load sharing), it is Communication and not Computational Complexity which acts as the main bottleneck for the running time of a typical parallel application. Thus, an algorithm which seeks to be “parallel-efficient” must reduce the number of operations which require Communication to an absolute minimum.

It has therefore been shown that, in their simplest and most obvious form, the major goals of a parallel algorithm are:

- **Correctness** – The algorithm must accomplish the goal for which it was designed, providing proper results as required by the application.
- **Computational Efficiency** – The Computational Complexity of the algorithm (usually expressed in “Big-O” notation) should be as minimal as possible while still preserving the correctness of the algorithm.
- **Simultaneous Workload (or “Load Balance”)** – The operations necessary to ensure correctness should be divided as evenly as possible amongst all processors allocated to the application.
- **Minimal Communication** – The amount of inter-processor Communication operations should be kept to the absolute minimum necessary to ensure correctness of the algorithm.
Developed at the University of Washington, ZPL is a parallel programming language whose acronym
doesn’t actually stand for anything. It follows Miranda-2 style syntax which will be featured throughout the
document and is the foundational language through which my research has been performed. ZPL provides three
major functionality differences from sequential programming languages which make creating parallel programs
more convenient and easier to understand.

### ARRAY-BASED COMPUTATION

In parallel computations, singleton values are rare. This is largely due not only to the nature of parallel
problems, but also due to the mindset necessary to tackle them. For example, consider a typical “FOR” loop which
increments a counter upon iteration:

```plaintext
for count := 0 to sentinel do
    something();
end;
```

For a sequential application, this makes sense. However, in the parallel world, each processor would have
to keep its own local version of the “count” variable, to avoid constant Communication at each step. This means
that “count” can no longer be considered a singleton value, but instead may be considered a 1-dimensional array
with as many entries as there are processors. ZPL performs this type of conversion automatically, so that the
logical representation of count is preserved and the physical implementation complexity is hidden (this is actually a
type of SCF).

ZPL further abstracts the notion of converting singletons into arrays through the use of “Regions”
depicted in Figure 1.

1 Figure courtesy Chamberlain, 2001.
itself has a sub-region, “TopRow”, which is a single-dimensional Region spanning the top row of “R”. Any changes made to elements in the “TopRow” region also affect the corresponding elements in “R” and “BigR”.

This makes parallel operations easier because if the author wishes to specify an operation to be applied to a set of data spanning multiple processors, the author instead can simply specify that operation in terms of Regions, and ZPL will handle the physical implementation. As an example of this, consider Figure 1.b. In this image, three variables are declared of size and dimension “BigR”; these variables are then used in a simple addition operation in Figure 1.c across Region “R”. The effect of this operation is that all of the values in Region “R” of B are added with all of the values in Region “R” of C and are placed into the corresponding cells in Region “R” of A.

Some arrays have special meaning. Specifically, an array named “Index<N>” where <N> refers to a natural number is pre-defined by ZPL to contain the Natural numbers in the Nth dimension.

![Index arrays](image)

Figure 2

As depicted in Figure 2.a, the “Index1” region contains the Natural numbers in the row dimension, while “Index2” contains the Natural numbers in the column dimension. As will be shown in more detail, these pre-defined arrays are useful tools for creating dynamic parallel algorithms.

ZPL supports all of the basic numerical operators on arrays as well as a few special operations. Only the four special array operations integral to my research are discussed.

**FLOODS AND REDUCTIONS**

Since the majority of data manipulation in ZPL is applied to arrays, there are built in operations for constructing arrays out of other arrays that may have different dimension or size specifications. One of these types of operations is known as “Flooding” and is depicted in Figure 3.a.

---

2 Figure courtesy Chamberlain, 2001

The Flood operation assigns information to the encapsulating Region that is acquired by replicating the information in the secondary Region. For example, in Figure 3.a, the encapsulating Region is “R” and the secondary Region is “TopRow”, thus, the statement:

\[ [R] \ A := \gg[\text{TopRow}] \ B; \]

Results in the Region “R” of variable A being set on a row by row basis to be equivalent to the “TopRow” Region of variable B. It is important to note that the Flood operation does not actually replicate the data, but is instead a purely logical operation. Figure 4 provides a mathematical representation of this Flood operation:

\[ A_{i,j} \leftarrow B_{i,j}, \forall (i,j) \in \mathbb{R} \]

Another type of instruction provided by ZPL which constructs arrays out of differently defined arrays is the “Reduction” operation. A reduction constructs an array whose size and dimension are defined by the encapsulating Region and whose information is based on the secondary Region and an “aggregate” function. In Figure 3.b, the aggregate function is “sum” (as indicated by “+<<”)\(^5\), thus the statement:

\[ [\text{TopRow}] \ A := +<<[R] \ B; \]

Results in the Region “TopRow” of variable A being set on a column by column basis to be the sum of all elements in the corresponding column in Region “R” of variable B. Figure 5 provides a mathematical representation of this Reduction operation:

\[ A_{i,j} \leftarrow \sum B_{k,j}, \forall (i,j) \in \text{TopRow}, \forall (k,l) \in \mathbb{R}, \text{ such that } l = j \]

---

\(^4\) Figure courtesy Chamberlain, 2001.
\(^5\) ZPL provides aggregate functions for sum, product, minimum, maximum, bitwise-AND, bitwise-OR, and allows for user-defined aggregates.
As may be seen from the prior examples, the Flood and Reduction operations are practically inverses of one another. Flooding expands information through logical replication, and Reduction contracts information via an aggregate function.

**SCANNING**

ZPL provides an operation known as “Scanning” which is similar to the Reduction operator (discussed prior) in that it is dependent on an aggregate function. The difference is that instead of contracting information based on the aggregate function, the Scan operation outputs a result which retains the dimensions and size of the original information, but consists of the progress of the aggregate (instead of the result, which is output by a Reduction) at each step.

For example, consider the array B defined over a Region “R”:

```
1 1
1 1
```

Scanning B over Region “R” using “sum” as the aggregate function and assigning the result to an array A would have ZPL syntax:

```
[R] A := +/|B;
```

And would result in the following output:

```
0 1
2 3
```

To achieve this output, a “tally” is initialized to 0 and the operation walks through the array B in row-major order, applying the aggregate function between the tally and the data in B, and storing the result in the corresponding cell in A. In this case, each element in array B is added to the tally and the tally’s value at each step is placed in A.

Note that this represents an “implicit” scan, meaning that the output of the aggregate is actually delayed by a single cell. To convert between an implicit scan and an explicit scan (where the outputs are not delayed), one must simply add the scanned array to the implicit scan’s result. The explicit result for the prior example is:

```
1 2
3 4
```

[R] A (Explicit)
REMAPPING

In many ways, the “Remap” operator is the most powerful array operation provided by ZPL, and the most difficult to understand. Unlike the previous instructions, Remap takes multiple arrays as parameters (one for each dimension of the destination Region) and the data in these arrays act as coordinates into the destination array. The information to be Remapped is then placed at the corresponding coordinate in the destination array.

As shown in Figure 6, array A is being Remapped by the data in arrays B and C. This is easiest to understand via the mathematical description, provided in Figure 7.

\[ A_{i,j} \leftarrow A_{B_{i,j}, C_{i,j}}, \forall (i, j) \in \mathbb{R} \]

Figure 7

As an example of the Remap operation, consider arrays A, B and C defined as follows:

\[
\begin{bmatrix}
1 & 2 \\
3 & 4
\end{bmatrix}
\quad \begin{bmatrix}
1 & 1 \\
2 & 2
\end{bmatrix}
\quad \begin{bmatrix}
1 & 2 \\
1 & 2
\end{bmatrix}
\]

Taken element by element in row-major order:

1. The first cell of array A will be remapped (moved) to row 1 (as defined in array B), and column 1 (as defined in array C).
2. The second cell will be remapped to row 1, column 2
3. The third cell will be remapped to row 2, column 1
4. The fourth cell will be remapped to row 2, column 2

The resultant array is thus:

\[
\begin{bmatrix}
1 & 3 \\
2 & 4
\end{bmatrix}
\]

6 Figure courtesy Chamberlain, 2001.
In this example, array B is equivalent to Index 1 and array C is equivalent to Index2. Note that by remapping A by [Index2, Index1], the resultant array is the transpose\(^7\) of A.

### AUTOMATIC LOAD SHARING

ZPL abstracts away the physical implementation of the parallel code, and thus is in charge of distributing the workload evenly amongst the processor pool. However, because ZPL is a Region-based language, the method through which it divides the work is intuitive and easy to understand.

\[\begin{array}{cccccccc}
G & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
0 & & & & & & & \\
1 & & & & & & & \\
2 & & & & & & & \\
3 & & & & & & & \\
\end{array}\]

**Figure 8**

Consider the 9-by-4 Region depicted in Figure 8. A single processor can execute an operation over this region by iterating through the elements of the array one at a time and performing the desired task. However, this type of execution pattern is not parallelizable, because each processor must start at the beginning of the array and traverse its entire contents, locking elements as necessary to avoid concurrency issues. As discussed prior, locking mechanisms and having multiple processors interact with the same data is contrary to the notion of parallel programming.

Instead, each processor is assigned a specific set of indices for all declared Regions (usually a set of columns or rows). Whenever an operation needs to be performed over a given Region, each processor executes the set of instructions for its assigned indices only (this includes storing the results in the target Region as well).

\[
\begin{array}{cccccccc}
\text{Processor 1} & \text{U} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\
\text{Processor 2} & & & & & & & & & \\
\text{Processor 3} & & & & & & & & & \\
\text{Processor 4} & & & & & & & & & \\
\end{array}\]

**Figure 9**

Figure 9 depicts the same 9-by-4 Region color-coded by processor assignment. Notice that each row is assigned to a specific processor and any operations over this Region will be executed in concert by all four processors, each working on their respective row.

---

\(^7\) The transpose of a matrix A (or 2-dimensional array) is the matrix B in which all rows have been converted to columns and all columns have been converted to rows.
**WYSIWYG COMMUNICATION MODEL**

Because minimizing Communication is at the heart of increasing performance for parallel algorithms, ZPL provides a “What-You-See-Is-What-You-Get” Communication model for its provided operations. In other words, only specific operators are permitted to utilize inter-processor Communication, and all other operators are guaranteed to not cause Communication. This means that the ZPL author can effectively control the number of Communication steps in the algorithm by limiting the usage of these operators to when they are absolutely necessary. Of the instruments discussed prior and those utilized for my research, the only Communication-prone operator is Remap, and as will be shown, is used at a minimum.

**PRACTICAL EXAMPLE**

The following is an example of a parallel algorithm for a common task, intended to exemplify the foundational change in mindset necessary to craft parallel solutions.

**SUMMA MATRIX MULTIPLICATION**

Matrix multiplication is a fairly common and trivial task. As depicted in Figure 10, given two matrices (expressed as 2-dimensional arrays), their product can be computed by performing the dot product\(^8\) for every row-column pair, and storing the result in their cell of intersection.

\[
\begin{array}{ccc}
1 & 2 & 5 & 6 \\
3 & 4 & 7 & 8 \\
\end{array}
\begin{array}{ccc}
1 \times 5 + 2 \times 7 & 1 \times 6 + 2 \times 8 \\
3 \times 5 + 4 \times 7 & 3 \times 6 + 4 \times 8 \\
\end{array}
\begin{array}{c}
5 + 14 \\
15 + 28 \\
\end{array}
\begin{array}{c}
6 + 16 \\
18 + 32 \\
\end{array}
\]

*Figure 10*

However, this algorithm is not a good parallel solution because there exists no segmentation of rows or columns in either the source or destination matrices which would allow multiple processors to compute the result in parallel without locks or Communication. The SUMMA Matrix Multiplication algorithm is designed to be highly parallelizable and makes use of a heretofore unmentioned parallel programming technique known as “Problem Space Promotion” (PSP).

**PROBLEM SPACE PROMOTION**

PSP is a technique in which the goal is to *increase* the Computational Complexity of an algorithm in such a way as to make the workload more easily separable. This is contrary to typical sequential coding styles because increasing the Computational Complexity goes against one of the two major sequential goals (discussed prior). Indeed, while parallel algorithms still wish to keep the Computational Complexity at a minimum, keeping the number of Communication steps low is far more important. For the preceding example, the problem space is

\(^8\) To perform the dot product between two vectors, simply multiply the corresponding values together and sum up the results. Ex: \(<a, b> \text{ dot } <c, d> = a*c + b*d\)
“promoted” by the SUMMA Matrix Multiplication algorithm by replacing the single dot product between the two 2x2 matrices:

\[
\begin{array}{cc}
1 & 2 \\
3 & 4 \\
\end{array}
\begin{array}{cc}
5 & 6 \\
7 & 8 \\
\end{array}
= \begin{array}{cc}
5+14 & 6+16 \\
15+28 & 18+32 \\
\end{array}
\]

Figure 11

With two array-products between two new 2x2 matrices (created by Flooding column-row pairs):

\[
\begin{array}{cc}
1 & 1 \\
3 & 3 \\
\end{array}
\begin{array}{cc}
5 & 6 \\
5 & 6 \\
\end{array}
= \begin{array}{cc}
5 & 6 \\
15 & 18 \\
\end{array}
\]

\[
\begin{array}{cc}
2 & 2 \\
4 & 4 \\
\end{array}
\begin{array}{cc}
7 & 8 \\
7 & 8 \\
\end{array}
+ \begin{array}{cc}
0 & 0 \\
0 & 0 \\
\end{array}
= \begin{array}{cc}
14 & 16 \\
28 & 32 \\
\end{array}
\]

Figure 12

This promotion has two effects. The first is that each individual array-product can be computed without knowledge of the other, meaning that different processors can work on their respective computations without the need for Communication or locks. Secondly, the result of each processor’s computations no longer represents the entire solution. Instead, the results must be aggregated, in this case by summing up the two output arrays (since these arrays can be treated as Regions, this aggregation is also load-shared across both processors).

It should be noted that as the size of the matrices grow larger, more work becomes available for the processing pool. For the SUMMA algorithm, the running time will continue to decrease as more processors are added until the number of processors is equal to the size of the largest dimension of the matrices being multiplied (i.e. when multiplying a 10x10 matrix by another 10x10 matrix, the running time will continue to decrease as processors are added until there are 10 total processors in the pool). This is a sign of good parallel algorithm construction because the running time will never increase as more processors are added to the available pool, but will instead plateau when there is simply no more work to be done.

**APPLICATION**

Parallel algorithms require multiple independent processors to be efficient. Interestingly, recent product releases from all of the major CPU manufacturers (most notably Intel and AMD) have shown that multi-core designs are the future of processor technology. As more cores come packaged onto the same chip, average home computers will become more capable than ever to make use of parallel programs, and, if the trend continues, parallelizable algorithms may eventually replace sequential algorithms as the primary executable medium. It was for these reasons that I decided to study parallel computations, and apply the lessons learned toward solving a real-world problem.

One of the major research subjects for modern parallel computing is the problem of Gene-DNA sequencing, and crafting an entirely new and efficient parallel solution has been the core of my research. However, before discussing my research on the subject, it is necessary to introduce the problem itself.
The problem that Gene-DNA sequencing algorithms seek to solve is to rapidly locate a smaller string of “nucleotides” (referred to herein as the “search string”) within a larger string of nucleotides (the “main string”), with the added level of complexity that the search string is not necessarily contiguous within the main string. That is to say, the search string may be spread out across multiple sections of the main string, with the only placement requirement being that it lies in sequential (left to right) order.

Each nucleotide is a single character chosen from the four-letter alphabet: “A, T, C, G”. Using the established terminology, the search string represents a Gene, and the main string represents the target strand of DNA. By being able to quickly locate specific Gene segments within DNA strands, Biologists can compare the effects of Gene variations (and/or lack of presence) with the effects on the DNA’s host, which has and will likely continue to lead to untold medical procedures and knowledge.  

As an example of the task at hand, consider the following main string:

ATGATCATTAG

Any search string which consists of letters contained in the main string in sequential order (from left to right) is valid, whether the individual letters themselves are contiguous within the main string or not. Thus, the following is a valid search string:

GACATA => ATGATCATTAG

The underlined portions indicate one possible mapping of Gene nucleotides within the DNA. However:

TGAGG => ATGATCATTAG x

Is an invalid search string, because the main string does not contain three “G”s. Furthermore:

ATGGT => ATGATCATTAG x

Is also an invalid search string, because there exists no sequence of letters in the main string which has two “G”s between two “T”s. Although the set of valid search strings is limited, there exist an infinite number of invalid search strings.

As will be discussed in more detail, while the search string may be considered invalid, there are portions of it which can be found within the main string, and one of the constraints is that the sequencing algorithm must be able to report these portions.

9 A full description of the usages for locating a Gene within DNA is outside the scope of this document.
The primary complication of finding the Gene within the DNA is simply the size of the inputs. A human cell has roughly $6 \times 10^9$ nucleotides on average, and an average Gene has about 2000 nucleotides (North, 2005). Thus, the sheer volume of data means that the Gene-DNA sequencer must be efficient, fast, and highly scalable.

**INDUSTRY SOLUTION**

The current industry solution is entitled the “Smith-Waterman Local Alignment Algorithm”, which is essentially a brute force approach toward solving the problem. The basic idea behind the algorithm is to, given a Gene string of length ‘M’ and a DNA string of length ‘N’, construct an “Edit Graph” as depicted in Figure 13.

![Edit Graph](image)

The Edit Graph uses a complex set of heuristics to score each cell (represented as orange circles in Figure 13) based on the cells surrounding it and on the match characteristics between the DNA and Gene strings. After each cell is scored, a set of “Trace Back Sequences” are computed by finding all paths from the bottom right cell of the Edit Graph to the top left cell, summing up the scores of each cell in the path, and returning the corresponding strings whose sum is less than a variable threshold.

Figure 13

---

10 Figure courtesy North, 2005
The Trace Back Sequences represent the set of acceptable permutations of the search string within the main string, and is robust to missing, moved, or altered nucleotides.\footnote{Figure courtesy North, 2005}

\section*{INHERENT PROBLEMS}

The primary reason that the Smith-Waterman algorithm is only mildly parallelizable is that during the Edit Graph computation, each cell’s score is dependent on the score of the cells near it. This means that there exists no segmentation of rows or columns in the graph such that neither Communication nor locks are required, and the computations to determine the score of a specific cell cannot be performed until all of the requisite surrounding cells have already been scored. Additionally, the process of acquiring the Trace Back Sequences requires computations to be processed with knowledge about the entire Edit Graph, meaning that not only can there be no parallel-efficient segmentation of data, but that no portion of the Trace Back Sequence computation can take place until the Edit Graph is completely filled.

The Smith-Waterman algorithm uses Communication at each step and does not efficiently load-share the computations across its processor pool, making the algorithm unnecessarily slow.

\section*{SOLUTION}

I approached this problem with three distinct initiatives in mind. Given that my knowledge of Computational Biology is limited, the first initiative was that I would remove the Biological restraints on the problem. Instead of simply matching “A, T, C, G” nucleotides, my solution allows for both the search and main strings to be from an arbitrarily large alphabet. The second initiative was that I would not base my solution off of the existing Smith-Waterman algorithm, but would come up with an entirely unique approach to solving the

\footnote{\textsuperscript{12} For a full description on how the Smith-Waterman algorithm works, please see Topic 10: “Smith Waterman Local Alignments”, \textit{Nucleic Acid Genetics and Sequence Alignments}, North, 2005.}
problem. Lastly, the solution must be highly parallelizable such that as more processors are added to the processing pool, the runtime would decrease to match.

The pages to come detail, in a step-by-step fashion, the process of locating all sections of the search string within the main string. This algorithm borrows no portion of its methods from pre-existing solutions, and represents the final result of 12 months of study, and repeated trial and error.

## STEPS TOWARD A RESULT

The idea behind the solution is to use a recursive algorithm with a divide-and-conquer approach. The algorithm first searches the main string for the entire, contiguous search string, and then recursively cuts the search string in half as it fails to find it. This divide-and-conquer approach is decidedly different from the Insertion-and-Deletion approach that industry standard algorithms such as Smith-Waterman utilize, and, when combined with a fast searching mechanism, may yield results in a faster (amortized) time.

### PERFORMING THE SEARCH

Instead of performing a linear search for the sub-divided search string, the algorithm is optimized for use within multi-processor environments. Thus the first step in performing the search is the utilization of the PSP technique, described prior.

### STEP 1: PROBLEM SPACE PROMOTION

Given a main string:

```
HELOWORLD
```

And search string:

```
OWOR
```

“Promote the problem space” by turning a 1-dimensional string into a 2-dimensional array. This is accomplished by flooding the main string vertically, in the following manner:

```
Array 1:  H E L L O W O R L D
          H E L L O W O R L D
          H E L L O W O R L D
```

The main string is flooded a number of times equal to the length of the search string, which, in this case, is 4 characters long. Similarly, the search string must also be flooded, in order to promote its dimension to the 2d plane. However, before flooding, the search string is first transposed, to act as a column of letters (instead of a row). The search string is then horizontally flooded a number of times equal to the length of the main string, and thus results in the following:
STEP 2: COMPARISONS AND SHIFTING

Now that two \( M \times N \) arrays exist, where \( M \) is the length of the search string and \( N \) is the length of the main string, they can be compared easily across multiple processors. Each processor is assigned coordinate sections within both arrays, and performs an “equals” operation between the two. This load-shared operation yields a new matrix consisting of a ‘1’ in any position where the two matrices are equal, and a ‘0’ in any position where they are unequal. For the prior example:

\[
\begin{array}{cccccccc}
H & E & L & L & O & W & O & R & L & D \\
H & E & L & L & O & W & O & R & L & D \\
H & E & L & L & O & W & O & R & L & D \\
\end{array}
\]

\[
\begin{array}{cccccccc}
W & W & W & W & W & W & W & W & W \\
W & W & W & W & W & W & W & W & W \\
R & R & R & R & R & R & R & R & R \\
\end{array}
\]

\[
\begin{array}{cccccccc}
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
\end{array}
\]

**Figure 15**

This operation has the effect of creating a perfect diagonal line of 1’s whenever an exact match is encountered. As can be readily seen, the search string “OWOR” begins at index 4 of “HELLO WORLD” (assuming 0-based numbering), and this is mirrored in the binary result in Figure 15 by the diagonal line beginning at index 4. In order to properly recognize and interpret these matches programmatically, the resultant binary matrix must be re-arranged, and then reduced. This is accomplished in three parts.

Array 2:

\[
\begin{array}{cccccccc}
O & O & O & O & O & O & O & O \\
W & W & W & W & W & W & W & W \\
O & O & O & O & O & O & O & O \\
R & R & R & R & R & R & R & R \\
\end{array}
\]

The problem space is now promoted to a 2d plane.
PART 1: NUMBER THE ROWS

Starting with 0 and incrementing downward, number the rows of the resultant binary matrix as follows:

<table>
<thead>
<tr>
<th>Row</th>
<th>0 0 0 0 1 0 1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

PART 2: SHIFT BY ROW

Circular-left-shift each row of the matrix a number of times equal to its row number. Note that the top row gets shifted by 0 (it stays in place), which preserves the original index of a possible match.

<table>
<thead>
<tr>
<th>Row</th>
<th>0 0 0 0 1 0 1 0 0 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0 0 0 0 1 0 1 0 0 0</td>
</tr>
<tr>
<td>1</td>
<td>0 0 0 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>2</td>
<td>0 0 1 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>3</td>
<td>0 0 0 0 1 0 0 0 0 0</td>
</tr>
</tbody>
</table>

This shift is accomplished using the Remap operator, and thus may require Communication between processors. However, this is the only step in the core algorithm which requires the use of a Remap. Computing the Remap array for the shift is as follows.

COMPUTE THE ROW REMAP ARRAY

Since none of the cells are moving across rows (but instead simply moving within the same row), the row remap array is Index1.

COMPUTE THE COLUMN REMAP ARRAY

This is the more complex step as it requires knowledge of two search parameters, the starting index of the search (which begins at 1 and may change as the recursive steps examine different portions of the main string), and the length of the main string. The formula for computing the array is:

\[ Col := (Index2 - (Index1 - (start - 1)) - 1)) \mod strLength; \]

Where “Col” is the column remap array, “start” is the starting index of the search, and “strLength” is the length of the main string. Note that the enclosing region for this operation is the same size and dimension as the
MxN array created during the PSP stage. After adjusting for negative overflow, the resulting column remap array for this example is:

<table>
<thead>
<tr>
<th>Row</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

After the remap is completed, any diagonal lines have been straightened into columns.

**PART 3: COLUMN-WISE “AND” REDUCTION**

For every column, each element within that column is logically ANDed to produce a single row which consists of a 1 if and only if every element in its corresponding column was a 1.

<table>
<thead>
<tr>
<th>$R_{row}$</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$\&\leftarrow$

$R_{row}$ | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |

The resulting row indicates (with a 1) every position where there was a search-to-main string match. In order to extract the indices of these matches, the resulting row is multiplied by the $Index2$ array.

<table>
<thead>
<tr>
<th>Row</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Index2$:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>$Result$:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

At this point, the 0's are ignored, and the minimum value is extracted from the row, yielding the intended result, a match at index 4.
STEP 3: RECURSION AND MINIMUM TRACKING

If the search string is not found, it is cut in half. Then, the search algorithm must recur on the left half of the search string before continuing to the right half, performing the search routine and re-recurring as necessary. The only additional element to this is that in order to enforce the sequential-search-string constraint, any results yielded by a recursion on the left half of the string must enforce, to the recursion on the right half, that no matches are acceptable at an index less than or equal to the match index in the left half added to the left half match length.

RESULT OPTIMIZATIONS

The search method represents the core idea behind the algorithm, but is not guaranteed to be robust for any combination of missing, altered, or moved nucleotides. This section outlines optimizations to the search routine to accommodate these additional constraints.

RETRIEVING PERCENTAGE OF SEARCH STRING FOUND

It is useful for the user to be able to see what percentage of the search string was found within the main string, instead of only seeing results when the entire search string was found. This allows the search algorithm to be robust to missing, moved or altered characters. There are three steps to computing the percentage.

1. Keep track of the total number of located characters.
   a. Whenever a portion of the search string is found, add the length of the substring to a running total.

2. Since erroneous characters are skipped, an additional value must be stored with each result: the location of the result in the search string.
   a. Maintain a “search-location” value, which begins at 1.
   b. Every time a match is found add the length of the match to the search-location value.
   c. In addition to storing the location of the result within the main string, also store the search-location value.

3. Make sure that recursion continues even if an erroneous character is encountered.
   a. Whenever the length of the search string is 1, if the character is not found within the main string, return the current minimum search index + 1, so that the subsequent recursion step ignores the character.
   b. Add one to the search-location value to keep its value accurate.

After all recursion has stopped, the percentage of the string found is simply:

\[
\text{Percentage} := \left(\frac{\text{Search-Location}}{\text{searchLength}}\right) \times 100
\]

Where “Search-Location” is the search-location value and “searchLength” is the length of the search string.

The ability to output the percentage of the search string found is a feat that the Smith-Waterman algorithm cannot claim. It allows the user to ignore the complex scoring heuristics (which may change from implementation to implementation of the Smith-Waterman algorithm) and instead receive results based on a percentage threshold.
RETRIEVING ALL POSSIBLE SEARCH RESULTS

Using the discussed methods, the search algorithm will only yield the position of the first located instance of any given substring of the search string. In other words, if there are multiple matches of the search string within the main string, then only the match with the lowest index (closest to the start of the main string) will be returned. The following modification to the algorithm allows the user to retrieve all positions of a located substring, stored as an array of integers. The change is made after the AND-Reduction step; instead of merely plucking out the index of the first match, the indices of all matches are copied as a row into a separate array (which stores all of the matches for all substrings of the search string, in the order in which they were found). To do this, the relevant information is remapped to the front of the row, and then all non-zero elements of the row are then inserted into the result array. Note that while this step requires an additional remap, it is only necessary when the user requires ALL possible search results. This mode of search returns results that are similar to those of Smith-Waterman because all possible search-to-main result permutations are reported.

Let C refer to a row of search-to-main results after the AND-reduction step. The process to move all non-zero information to the front of the row, while maintaining their relative positions, is as follows:

1. \( C_{Scan} := +1/C \);
2. \( ColRemap := (C_{Scan} + C) \times C \);
3. \( C_{Reduction} := +<<C \);
4. \( ColRemap := ((ColRemap + 1) \times Index2) + C_{Reduction} - C_{Scan} \);

\textit{end;}

Taken step by step:

1. Construct an implicit plus-scan of C.
2. Convert this plus-scan to an explicit plus-scan \( (C_{Scan} + C) \) and then convert this explicit scan into a monotonically increasing sequence \( (\ast C) \).
3. Acquire the total number of relevant pieces of information by performing a sum reduction. Since each match is represented as a 1, the value of this reduction is equal to the number of matches.
4. For all non important information (values equal to 0):
   a. Retrieve their position in C: \( ((ColRemap + 1) \times Index2) \)
   b. Make sure all of their positions are larger than the important pieces of information: \( + C_{Reduction} \)
   c. Correct for any overflow to avoid out-of-bound indices: \( - C_{Scan} \)

This constructs the column remap array necessary to properly order the elements of the row. For the example, this matrix is:

<table>
<thead>
<tr>
<th>Row</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>0</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
This has the effect of moving the match (at the 5th element) to the front of the row so that it can be stored into the result array.

Note that this entire operation represents added Computational Complexity and Communication for memory efficiency and ease of interpretation. If, for every recursion yielding at least 1 match, the entire result row was stored in the result array, the additional remap steps would be unnecessary. However, this would make interpreting the results far more difficult and their storage more costly, since the entire row would be stored instead of merely the non-zero elements. While for this example storing additional 0s results in very little additional storage, if the main string were 6 billion characters long and the search string matched it in only 2 places, then this remap would drastically increase memory efficiency.

Additionally, this type of search allows for the user to not only see the percentage of the search string that was found, but also where they were found, a value the Smith-Waterman algorithm cannot return. This means that the user can look at the DNA directly and gather information about the nucleotides surrounding the Gene.

---

**CORRECTING FOR “MALICIOUS” INPUTS**

There are two very rare input cases in which the search algorithm (as described thus far) would return improper results.

As an example of the first case, given the main string:

```
ATGATCATTAG
```

And the search string:

```
ATATAT
```

The expected result would be the **100%** located string:

```
ATGATCATTAG
```

However, the algorithm would take the following steps:

1. Perform a full search.
   a. Search String: **ATATAT**
      i. **No Results**
2. Recursively divide the string into two segments of length three.
   a. Left Recursion Search String: **ATA**
      i. **No Results**
   b. Right Recursion Search String: **TAT**
      i. **No Results**
3. Recursively re-divide the segments.
   a. Left-most Recursion Search String: **AT**
      i. **String Found** at location 0
   b. Left Recursion Search String: **A**
The problem is that the algorithm always gives preference to characters that appear furthest to the left when sub-dividing strings of odd length. To fix this issue, the algorithm was modified such that, during left-side recursions, preference is given to the left side of odd-length strings, and, during right-side recursions, preference is given to the right side. This has the effect of allowing for a larger granularity of results near the center (where the mistakes are more likely to appear), with the larger pieces at the ends. This modification yields the following results:

Given the main string:

```
ATGATCATTAG
```

And the search string:

```
ATATAT
```

The expected result would be a **100%** located string.

The algorithm takes the following steps:

1. **Perform a full search.**
   
   a. **Search String:** ATATAT
      
      i. **No Results**

2. **Recursively divide the string into two segments of length three.**
   
   a. **Left Recursion Search String:** ATA
      
      i. **No Results**
   
   b. **Right Recursion Search String:** TAT
      
      i. **No Results**

3. **Recursively re-divide the segments.**
   
   a. **Left-most Recursion Search String:** AT
      
      i. **String Found** at location 0
   
   b. **Left Recursion Search String:** A
      
      i. **String Found** at location 3
   
   c. **Right Recursion Search String:** T
      
      i. **String found** at location: 4
   
   d. **Right-most Recursion Search String:** AT
      
      i. **String found** at location: 6

**Actual Result:** **100.00%**
The second case is that if a pattern of characters emerges at some point in the main string that are coincidentally the same as a set of characters in the search string that are out of order in the main string, then the characters will confuse the minimum index of recursion.

For example, given a main string of:

```
ANYTIMEISTIMEFORTACOSDONTYOUTHINK
```

With a search string of:

```
TACOTIMEISTIME
```

The best percentage match would be 71.42% by locating TIMEISTIME. The substring TACO should be ignored because it is a smaller substring than TIMEISTIME and locating both would violate the sequential-search-string constraint. However, what happens instead is the following:

1. Perform a full search.
   a. Search String: TACOTIMEISTIME
      i. No Results

2. Recursively divide the string into two segments.
   a. Left Recursion Search String: TACOTIM
      i. No Results
   b. Right Recursion Search String: EISTIME
      i. No Results

3. Recursively re-divide the segments.
   a. Left-most Recursion Search String: TACO
      i. String Found at location 16
   b. Left Recursion Search String: TIM
      i. No Results
   c. Right Recursion Search String: EIS
      i. No Results
   d. Right-most Recursion Search String: TIME
      i. No Results

Actual Result: 28.57%

To remedy this problem, two passes over the data are necessary. The first pass operates as normal, starting from the front of the main string, locating matches and adjusting the minimum index accordingly. The second pass starts from the back of the main string, and tests the recursive substrings in reverse order (i.e. it checks the right-most recursion first), and adjusts the maximum index accordingly. After both passes of the data have been complete, the higher result percentage is returned. This modification yields the following results:

Given a main string of:

```
ANYTIMEISTIMEFORTACOSDONTYOUTHINK
```
With a search string of:

TACOTIMEISTIME

The best percentage match would be 71.42\% by locating TIMEISTIME. The substring TACO would be ignored.

Perform normal-style recursion, yielding a 28.57\% find.

Perform reverse-style recursion:

1. Perform a full search.
   a. Search String: TACOTIMEISTIME
      i. No Results

2. Recursively divide the string into two segments.
   a. Right Recursion Search String: EISTIME
      i. No Results
   b. Left Recursion Search String: TACOTIM
      i. No Results

3. Recursively re-divide the segments.
   a. Right-most Recursion Search String: TIME
      i. String Found at location 9
   b. Right Recursion Search String: EIS
      i. String Found at location 6
   c. Left Recursion Search String: TIM
      i. String Found at location 3
   d. Left-most Recursion Search String: TACO
      i. No Results

Reverse style recursion yields a 71.42\% find.

Actual Result: max(28.57, 71.42) = 71.42\%

In order to guarantee that the highest percentage of the search string is located within the main string, two passes are necessary. However, this type of conflict happens with such low probability (and indeed the probability decreases as the size of the search and main string increase) that in most cases the algorithm can be run without executing the reverse search.

**SPEED OPTIMIZATIONS**

| ARRAY LOOKUP OPTIMIZATION |

According to the discussed methods, the algorithm will sub-divide the search string, perform a lookup into the main string, compare the two for equality, and then act as normal afterward. However, since the entire search string gets compared to the entire main string at the first iteration, subsequent sub-divisions and comparisons lead to duplicated work. Instead, before the sub-division recursion process is started, the entire search string is compared against the entire main string and the resulting bitmap is stored. During further recursive steps, when a
subset of the search string would normally be compared against the main string, the result is instead “looked up” in the previously stored bitmap.

Before this optimization, given a current search string of size M (“current” referring to the size of the search string at the relevant recursive step), and a main string of size N, the creation of the comparison array involves \((M \times N)\) steps which is repeated as many as \(\log_2 S\) times, where \(S\) is the size of the entire search string. The effect of this optimization is such that only one comparison is ever made, and it is always of complexity \((S \times N)\). Furthermore, the time it takes to perform such a comparison is no longer dependant on the number of recursive steps needed to locate the search string within the main string. Thus, highly complex searches will run at the same speed as mundane ones, and the comparison array can be entirely pre-computed!

To see a pictorial representation of this, consider the following main string:

<table>
<thead>
<tr>
<th>F</th>
<th>I</th>
<th>S</th>
<th>H</th>
<th>T</th>
<th>A</th>
<th>C</th>
<th>O</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>I</td>
<td>S</td>
<td>H</td>
<td>T</td>
<td>A</td>
<td>C</td>
<td>O</td>
<td>D</td>
</tr>
</tbody>
</table>

With search string:

<table>
<thead>
<tr>
<th>I</th>
<th>S</th>
<th>Z</th>
<th>O</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>S</td>
<td>H</td>
<td>T</td>
</tr>
</tbody>
</table>

Without this optimization, the algorithm would take the following steps:

1. Flood the search and main strings and compute the equality bitmap.

   | F | I | S | H | T | A | C | O | D | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
   | F | I | S | H | T | A | C | O | D | 0 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
   | F | I | S | H | T | A | C | O | D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
   | F | I | S | H | T | A | C | O | D | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

   ![Figure 16](image)

2. Check for a full match, when one is not found, recursively sub-divide both sides until matches are found.
However, the additional bitmaps that are created at each step are merely subsets of the original bitmap created during the first MxN comparison. Thus the search algorithm can take the following steps instead:

1. **Pre-Compute the MxN equality bitmap.**

   ![Figure 18](image)

   Figure 18

   At each recursive step, instead of sub-dividing the search string, sub-divide the pre-computed bitmap, and use it to check for search results.

   ![Figure 19](image)

   Figure 19

   This optimization results in a massive performance increase as it is both memory efficient and computationally efficient.
ADDITIVE OPTIMIZATION

It is the nature of this algorithm to locate the larger parts of the solution first, leaving smaller “fragments”, which may be part of the solution, as separate results. For example, a search for:

LIGHYLARTTOSEEWHTIESOFTHES

Within the main string:

THISISASLIGHTLYLARGERTOSEEWHTHAH THECAPABILITIESOFTHESYSTEMARE

Will likely yield the following results:

<table>
<thead>
<tr>
<th>Index</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>12</td>
<td>1</td>
</tr>
<tr>
<td>15</td>
<td>2</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
</tr>
<tr>
<td>24</td>
<td>1</td>
</tr>
<tr>
<td>25</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>45</td>
<td>3</td>
</tr>
<tr>
<td>48</td>
<td>7</td>
</tr>
</tbody>
</table>

Where “Index” refers to the index of the main string in which the portion of the search string corresponding to “Length” was found. For example, the match “Index: 9, Length: 3” means that the first 3 letters of the search string were found in the main string at index 9, while the next match “Index:12, Length: 1” means that the 4th letter of the search string was found at index 12 of the main string.

While these results are accurate, it would be nice to make them more concise, by increasing the size of the large result pieces. There are two optimization techniques used to accomplish this, and both may be used individually or in tandem. Note that both optimization routines are sequential algorithms that are post-computed after all searching has been completed, and neither is necessary (or correct) when searching for all possible matches.

1. Iterate over the results, adding each index to its corresponding length.
2. Check if this sum is equal to the next index.
3. If it is, remove the next index, and add it’s the next index’s length to the length of the previous index.
4. If it isn’t equal or the optimization for this result has been completed, continue to the next index, and perform 1-4 until reaching the end of the result list.

Using this optimization routine, the results would be compressed into the following, where bold lettering indicates Indexes/Lengths that were adjusted.
For the given search/main string, this happens to be the optimal result segmentation.

**“WIGGLE” OPTIMIZATION**

Another kind of fragmentation may occur when the main string contains many different subsets which match the same part of the search string. Thus, when a left-half recursion occurs, it may locate its target many times, and, by way of convention, pick the result with the lowest index (closest to the start of the string). This has the effect of possibly shortening a result when the right-half recursion performs its search, because a left-half match may be at the start of a right-half result. For example, when searching for:

```
FISHTACO
```

With a left-half search for “FISH” and a right-half search for “TACO”, within:

```
FISHANDTORTILLASMAKEFISHTACOS
```

The left-half search string is found in two places, at the very beginning of the main string and just before “TACOS” at the end. This is depicted in Figure 21.
Even though “FISH” will be found twice, the left-most location will be preferred, thus leading to the following result:

```
FISHANDTORTILLASMAKEFISHTACOS
```

Instead of the intended:

```
FISHANDTORTILLASMAKEFISHTACOS
```

In order to optimize away such outcomes, the left-half results are “wiggled” in an attempt to find a larger, more contiguous result. This is accomplished in the following way:

1. For a given result, “slide” the resultant string to the front of the next result, and check to see if this new string matches the main string at the same indices.  
2. If it does, remove the old result, decrement the index of the next result by the length of the old result, and increment the length of the next result by the length of the old result.  
3. If it doesn’t match the main string or once the optimization for a given result is complete, move on to the next result until all results have been “wiggled”.

To perform the “slide” operation, another set of remap arrays must be computed. However, since this optimization algorithm only operates on the results of the larger search, the amount of Communication is far lower than if the remap were applied during each recursive step.

To compute the “slide” operation’s remap arrays:

1. Compute the row remap array
   a. Since none of the cells are moving across rows (but instead simply moving within the same row), the row remap array is simply Index1.
2. Compute the column remap array, using the following formula:

   \[
   \text{Col} := (\text{Index2} + ((\text{nextIdx} - \text{index}) - \text{length})) \mod \text{strLength};
   \]

   Where “Col” is the column remap array, “nextIdx” is the index of the next match, “index” is the index of the current match, “length” is the length of the current match, and “strLength” is the length of the main string.

Once the remap has completed, the given portion of the search string corresponding to the result is moved to be adjacent to the next result portion of the search string so that it can be compared against the main string. This optimization drastically decreases the need to perform the reverse-traversal search as outlined in the “Correcting for ‘Malicious’ Inputs” section, as it will correct for many of the errors caused by those inputs.

RESULTS

Unfortunately, it is prohibitively expensive to test this algorithm against the industry standard on any fair basis. Doing so would require access to the types of server clusters that currently run the Smith-Waterman algorithm for professional means, as well as an extremely optimized implementation of this program (the Smith-Waterman applications have been hand-optimized at the assembly level for years). However, speed tests were done on two multi-processor systems to show that the algorithm does indeed conform to good parallel design, and
gets (much) faster as more processors are allocated to the available pool. A brief description of the two systems’ specifications follows.

<table>
<thead>
<tr>
<th>System</th>
<th>Specifications</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTU</td>
<td>URL: attu.cs.washington.edu&lt;br&gt;Processing Power: 4 processors each with 1 context, arranged in a 2x2 grid&lt;br&gt;Operating System: Linux</td>
</tr>
<tr>
<td>SUNFIRE</td>
<td>URL: sunfire.cs.washington.edu&lt;br&gt;Processing Power: 8 processors each with 4 contexts, arranged in an 8x1 grid&lt;br&gt;Operating System: Solaris</td>
</tr>
</tbody>
</table>

Even though these servers only have a limited number of physical processors, it is possible to “pretend” that there are more available processors by simply increasing the workload passed to each processor (and passing it to them under different kernel threads or “contexts”).

Due to memory restrictions, the largest test size that ATTU would support was a 1,000 character search string and a 15,000 character main string. Both systems were timed using these exact same parameters and the search was performed 25 times, adding one processor to the working pool at each step (ATTU supports a maximum of 25 virtual processors). The results are shown in Figure 22; the Y-Axis represents the running time in seconds, and the X-Axis is the grid configuration of the processors.

![Figure 22](image-url)
The shape of the graph indicates that the algorithm does indeed grow faster as more processors are added, until eventually the running time plateaus. However, there are jitters in the graph in the “All Values” plot, where, upon adding an additional processor, the running time actually increased! The reason for this is that ATTU’s processors are arranged in a 2x2 grid, and any grid configuration which does not evenly split into this orientation will cause some of the processors to carry an extra workload. In the case of prime configurations (that is, where the number of processors being used is a prime “P” and therefore the only available configuration is Px1), one of the processors ends up doing far more work than the others, and the running time suffers because of this. In the “Primes Excluded” plot, it is plain to see that most of the jitters have gone. The graph’s remaining upward spikes are due to grid arrangements in which one dimension is much larger than the other (i.e. 7x2 or 11x2), again causing more work to rest on some processors than on others. Removing any arrangements in which the difference between grid dimensions is larger than 2, yields the “Semi-Symmetric” plot, and removing all grid arrangements in which the dimensions are not equal yields the “Symmetric” plot. Both the Symmetric and Semi-Symmetric plots are monotonically decreasing (until the 5x5 processor case, which was ATTU’s breaking point).

These same tests were run on the SUNFIRE system, in which each individual processor is slower than those on ATTU, but are more numerous. The “SUNFIRE” plot shows that even though the running time on a single processor may be much slower, the increased number of physical processors leads to an overall decreased running time. Note that because the SUNFIRE system is arranged in a prime grid (8x1), prime grid arrangements are the most efficient, and thus the fastest.

<table>
<thead>
<tr>
<th>System Name</th>
<th>Running Times</th>
</tr>
</thead>
<tbody>
<tr>
<td>ATTU</td>
<td>Slowest: 9.31 seconds on a 1x1 grid</td>
</tr>
<tr>
<td></td>
<td>Fastest: 3.84 seconds on a 4x4 grid</td>
</tr>
<tr>
<td></td>
<td>Running Time Decreased by a factor of 2.42 for 3 additional processors</td>
</tr>
<tr>
<td></td>
<td>(15 additional threads)</td>
</tr>
<tr>
<td>SUNFIRE</td>
<td>Slowest: 20.6 seconds on a 1x1 grid</td>
</tr>
<tr>
<td></td>
<td>Fastest: 2.09 seconds on a 23x1 grid</td>
</tr>
<tr>
<td></td>
<td>Running Time Decreased by a factor of 9.86 for 7 additional processors</td>
</tr>
<tr>
<td></td>
<td>(22 additional threads)</td>
</tr>
</tbody>
</table>

These staggering numbers lead me to use the more powerful SUNFIRE system to test a much larger data set over an increased number of processors to see if the trend continued. For the following tests, the search string was 2,000 characters, and the main string was 60,000 characters. The search was performed 41 times, adding one virtual processor (thread) to the available pool at each step. Figure 23 portrays the results.
Even though there was a drastic decrease in the overall running time by adding 40 additional threads to the task, it doesn’t represent much of a speed-up over adding 22 threads in the previous test case. This is likely due to the fact that the system only has 8 physical processors. If a system with more physical processors were available, all indications show that the running time would continue to decrease.

Following are graphs for each system depicting the run time speed-up per processor as a factor of the sequential running time, computed by dividing the parallel running time at each point by the sequential (single-processor) running time. Also included for each system is the efficiency of adding the additional processor with respect to the overall running time, computed by dividing the speed-up factor by the number of processors utilized. The graphs for the two SUNFIRE tests were virtually identical, and therefore only one is provided. Note that these graphs only make sense for physical processors.
The graphs are as expected, the Speed-Up per processor has a strong positive trend and the efficiency per processor has a negative trend. For ATTU's four processors (each with a single context), the efficiency drops fairly quickly. However for SUNFIRE, where each of the eight processors has four contexts, the efficiency remains much closer to 1 (and the speed gained per processor is much higher) even while all physical processors are being utilized, further showing that the algorithm benefits greatly from (and makes efficient use of) additional physical processors.
IMPACT AND PROPOSED USAGE

Should this algorithm prove faster than the industry standard, the impact on the field of Computational Biology would be staggering. Gene-DNA sequencing is amongst the most common of usages for parallel computers and the Computational Biology community has spent millions of dollars researching optimization methods for the Smith-Waterman Local Alignment algorithm for use in multi-processor environments (North, 2005). However, the problem is that while the program may be made parallel, the algorithm itself is not an inherently parallel solution.

One currently existing parallel solution to this problem is “BLAST”, a method through which the steps leading up to the usage of the Smith-Waterman algorithm can be processed in parallel. However, the BLAST algorithm introduces uncertainty in the results, as it trades accuracy for speed. The proposed usage for this research is to replace the application of the Smith-Waterman algorithm in BLAST executions, using the versions of the search that are not guaranteed to find all results. By doing this, a small amount of accuracy is traded for a vastly expedited search time, which is a trade that users of BLAST have already implicitly made.\(^\text{13}\)

SUMMARY

After months of learning parallel programming techniques under the instruction of Professor Larry Snyder, I decided to apply them toward the problem of locating a gene within a DNA strand. To accomplish this task I used a parallel programming technique known as “Problem Space Promotion”. Essentially, the goal is to increase the computational complexity of a problem in such a way as to make it easier to sub-divide the problem space over multiple processors. The end result is that even though the total number of computational steps is larger, the number of steps each individual processor performs becomes smaller, thus decreasing the overall running time. I applied the technique to a recursive, divide-and-conquer style approach to solving the problem, and ended up with a solution who’s running time decreases dramatically as more processors are allocated to the task and which uses minimal inter-processor communication. Utilizing the PSP method is a decidedly new, complex and powerful means of solving the problem and exposes tremendous possibilities for further exploration.

ACKNOWLEDGEMENTS AND AWARDS

This research would not have been possible without my adviser and instructor, Professor Larry Snyder of the University of Washington’s Computer Science and Engineering department. Additionally, I would like to thank Professor Vladimir Chaloupka of the University of Washington’s Physics department for first introducing me to this problem, and Steve Henry at Windows Internals, Microsoft for aiding with some of the design decisions.

This research has earned an Honorable Mention for the 2007 Computing Research Association Outstanding Undergraduate Research Award, and given that there is only a single winner and 10 honorable mentions (per gender), this was quite an honor.

REFERENCES


\(^{13}\) For a complete description of the BLAST algorithm, see North, 2005.