High-Performance Direct Pairwise Comparison of Large Genomic Sequences

Christopher Mueller, Mehmet Dalkilic, and Andrew Lumsdaine
Open Systems Lab/Computer Science Department, Indiana University
Bloomington, IN 47405
{chemuell, dalkilic, lums}@indiana.edu

Abstract

Many applications in Comparative Genomics lend themselves to implementations that take advantage of common high-performance features in modern microprocessors. However, the common suggestion that a data-parallel, multithreaded, or high-throughput implementation is possible often ignores the complexity of actually creating such software. In this paper, we present a data-parallel algorithm for a classic comparative genomics algorithm, the dot plot, along with a multiprocessor extension. For large genomic comparisons, these new algorithms achieve speedups of up to 14.4x over the sequential version. This speedup introduces the opportunity of performing full pairwise comparisons on entire genomes on a much larger scale than previously possible. We also present the experimental, model-driven approach used to develop the algorithm that allowed us to carefully study and evaluate implementation options and fully understand the parameters effecting its performance.

KEYWORDS dot plot, data-parallel, pairwise comparison, sequence alignment, vector processor, Altivec, high-performance computing, comparative genomics

1. Introduction

The *dot plot* algorithm [9] is one of the oldest computational tools for comparative genomics. It creates a pairwise comparison between two sequences and renders the results as a dot-matrix (Figure 1). A dot-matrix for two sequences Q and S is simply a grid with the presence of a point at position $p = (i, j)$ if the k-tuple beginning at i^{th} position of S and the j^{th} position of S coincide.

For years, the quadratic running time dot plot was acceptable as most available sequences were short. Over the last decade, however, sequence-alignment tools such as BLAST [1] and FASTA [12] have gained favor for their fast and accurate results for finding "best" alignments between a short query sequence and larger sequence database.

This approach is effective for studying individual genomic components, but fails to capture the complete relationship between larger sequences. Recently, with the sequencing of more full genomes, the need for tools to study this relationship has become an active area of research, with a focus on providing whole-genome alignments [5,6]. However, due to the intractable nature of finding a best alignment between large sequences, these algorithms must rely on filtering techniques and can fail to capture the complete relationship between large sequences. The dot plot algorithm is still the only algorithm for computing a full direct pairwise comparison between two sequences.

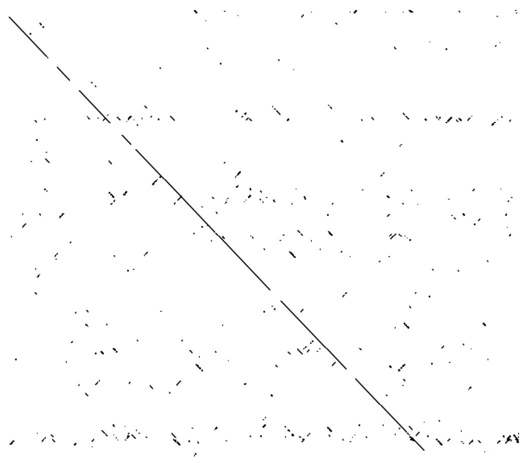

Figure 1 A Dot Plot. Common regions between sequences as appear long diagonals. Short, repeated sequences appear as horizontal features. For small window values, inverted sequences appear as reverse diagonals.

Along with the increase in the size and availability of genomic data, commodity microprocessors have also expanded to include high-performance feature sets. For example, all common desktop processors include a vector processing unit that allows single-instruction multiple-data (SIMD) level parallelism. These vector processors are commonly used in media applications where large amounts of streaming data must be processed quickly. In addition to different processing units within a single processor, many desktop computers also include multiple processors and all major operating systems provide support for thread-based shared-memory parallel applications.

In this paper, we demonstrate how to apply these process features to the dot plot algorithm to compare large sequences.

2. Related Work

Data-parallel versions of some common bioinformatics algorithms exist. Apple/Genentech BLAST, AGBLAST[2], uses Apple's Velocity Engine to enhance BLASTN alignments. For large nucleotide word sizes, AGBLAST attains a 5x speedup over the standard implementation. However, the results are highly dependent on the word size and the improvement for more sensitive searches is not as dramatic.

[13] presented a data-parallel version of the Smith-Waterman [14] dynamic programming algorithm for finding optimal local alignments. The core of Smith-Waterman is similar to the dot plot algorithm, but narrows the comparison space by only following paths that will lead to good local alignments. Using the MMX vector unit in Intel processors [10], a 6x speedup was achieved.

Erik Lindahl at Stanford has reportedly [7] implemented an Altivec enhanced version of HMMER that is 30% faster than the standard version.

3. The Dot Plot Algorithm

The basic dot plot algorithm compares every character in one sequence against every character in another sequence, placing a dot in the resulting grid where the characters match. When visualized, the matrix exposes areas of similarity between the sequences as continuous diagonal lines (Figure 1). More complicated patterns inform biologists of other similarities between the sequences, such as inversions, repeated sequences, and areas of low complexity.

The basic algorithm is adequate for comparing short DNA and medium sized protein sequences. However, due to the small size of the DNA alphabet, the matrices often become noisy and features are lost. The solution is to compare the sequences using a sliding window that compares w consecutive characters in each sequence. If the number of matches is greater than a stringency threshold *strig*, a dot is added at the beginning of this window. This filters out areas of insignificant matches and highlights longer ones. The DOTTER program [15] takes this approach one step further by storing the number of matches in each window and rendering them as grayscale values, showing more detail than the basic windowing algorithm. Window and stringency values are set based on the types of sequences (DNA/protein) and the amount of detail required. High values will show only significant long matches. Additionally, inversions only appear when then window size is 1. A common method for showing inversions using windows is to reverse one sequence and perform the comparison a second time. Window sizes of 15-30 are common for DNA with the stringency usually set at about 60% of that.

Given two sequences, q and s, a window size w and stringency value *strig*, the naïve implementation of the algorithm has a running time of $O(|q||s|w)$. The w term is important. For large sequences, the quadratic time computation is feasible if w is very small (1 or 2). However, a realistic window size makes the computation impractical. For instance, given two medium size bacterial genomes with 3 Mbp and a computer capable of comparing 100 Mpbs per second, the naïve algorithm with w=1 could compare the sequences in 25 hours. Each incremental addition to the window sizes will increase the computation by about a day.

An improved version [11] of the dot plot algorithm removes the runtime dependency on w. The new algorithm first computes a score vector for each letter in the alphabet against the horizontal sequence q. For a sequence with an alphabet $\sum$, this generates $|\sum|$ score vectors, each of length $|q|$. Then, starting with a running score vector r of length $|q|$ initialized to 0, for each character c in the vertical sequence s, it adds the entire score for c to r. r is shifted to the right by 1 for each iteration so that the scores accumulate along the diagonals. To handle the window size, at each step the score vector for the character in s from w iterations back is *subtracted* from r. Thus, r only contains the current scores in row i of the matrix. After each step, r is scanned and any positions such that $r[j] > strig$ and $0<=j<=|q|$ are recorded in the resulting dot plot.

This version has a running time of $O(|q||s|)$, but increases the space requirements to $|\sum||q|$. However, this is manageable for all genomic comparisons, where $|\sum_{DNA}|$ = 4. Although the largest genomes contain more than 3 billion base pairs, they can be easily broken into manageable chunks.

4. Two Parallel Dot Plot Algorithms

To take advantage of vector processors and multiple processors in commodity systems, we now present two extensions to the dot plot algorithm. Because of the complexity of comparing protein sequences with their larger alphabet sizes, we focus on large DNA comparisons.

Data-parallel processors allow the same operation to be applied in parallel to all the values stored in a vector register. Vector registers can be segmented to contain most common data types: chars, shorts, ints, and floats. The size of the register and the size of the data type determine how many elements are operated on at once. For instance, a 128-bit register can hold 16 8-bit chars, 8 16-bit shorts, or 4 32-bit ints. Common operations available in vector processors include addition (`vec_add`), comparison (`vec_ge`, `vec_any_lt`), and permutation for rearranging elements in vectors (`vec_perm`).

To support data-parallelism, we exploit the fact that windows for comparing DNA sequences are usually small (<30) and rarely ever larger than 100. Assuming every character in the window matches and each match is

```
DPDOTPLOT(qScores, s, win, strig):
  # Parameters:
  #  qScores: the precomputed score vectors for
  #    [A,C,G,T] against the entire sequence q
  #  s: the vertical sequence
  #  win: the comparison window
  #  strig: the stringency

  # Process each diagonal in the matrix, using
  # 16-element vectors along q.
  for each vector diagonal D:
    # Zero the running score vector
    runningScore = vector(0)

    # For lower triangle vector diagonals,
    # accumulate the score for the start of
    # the vector using the standard processor
    runningScore = ProcessDiagonalHead(D)

    # Accumulate the score for the diagonal
    for each char c in s:
      # Load the score vector for character c
      # from q's precomputed score vectors and
      # add the score to the running score
      score = VecLoad(qScores[c])
      runningScore = VecAdd(score, r_score)

      # Subtract the values outside the
      # window from the running score.
      if index(c) > win:
        delChar = s[index(c) - win]
        delscore = VecLoad(qScores[delChar])
        runningScore = VecSub(score, delscore)

      # Compare the running score against the
      # stringency
      if VecAnyElementGte(runningScore, strig):
        # Unpack and save the results
        scores = VectorUnpack(runningScore)
        for each score in scores > strig:
          Output(row(c), col(score), score)
        end for each score
      end for VecGte()
    end for each c

    # For upper triangle vector diagonals,
    # accumulate the score for the end of
    # the vector using the standard processor
    runningScore = ProcessDiagonalTail(D)
  end for each D

end DPDOTPLOT
```

Listing 1 The data-parallel dot plot algorithm

scored as 1 and each mismatch as 0, the maximum value the score vector holds for each position is w, the size of the window. Thus, the size needed to store a single score is $s_n = \lfloor \log_2 w \rfloor$. Because score addition will not overflow a storage element and subtraction will not cause underflow, arbitrarily sized score elements can be used, leading to a maximum increase in the number of comparisons made per operation to be $n_{cmp} = s_{reg} / s_n$. For $w = 30$ and $s_n = 5$, a 128-bit register could hold 25 score values, providing a theoretical 25x increase in performance. In general, however, it is more practical to work in increments already supported by the processor.

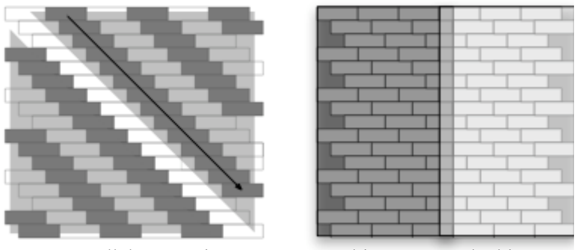

Figure 2 The Data-parallel and Multiprocessor Dot Plot Algorithms. The data-parallel algorithm (left) processes each 16-element wide diagonal in the matrix, starting with the upper triangular portion first. The multiprocessor algorithm (right) breaks the matrix into two column blocks. The right block overlaps the left block to accumulate the window score before it begins processing.

Rather than accumulating the running score over the complete length of q at each iteration, our algorithm operates on a score vector of length n_{cmp} and computes an entire n_{cmp} wide vector diagonal band of the dot plot. For instance, using vectors that hold 16 chars, n_{cmp} would be 16 and we would compute a 16 character wide vector diagonal in one pass. This approach has two practical advantages. First, the score vector can be held in a register for the entire duration of the computation and does not need to be stored in memory, as would be the case if the entire score vector were accumulated. Second, the resulting matrix is serialized along each diagonal, simplifying post processing and rendering.

The vector algorithm is shown in Listing 1 and illustrated in Figure 2. For each 16-element vector diagonal in the matrix, it computes the running score for the diagonal from q's score vectors. At each step, it performs a vector comparison against the stringency value and if any elements are greater than or equal to it, their location and value are saved. The beginning (lower triangle) and end (upper diagonal) of the vector diagonals that are less than the width of the vectors are processed using the standard processor.

To extend the algorithm to support multiple processors, we block the resulting matrix by dividing q into equally sized column blocks for each processor. The right hand side block must overlap the left hand block by w characters to allow the window score to accumulate (Figure 2).

The final important feature of both parallel algorithms is the data structure used to store the resulting matrix. Even for small bacterial genomes on the order of 3

Mbp, the complete comparison matrix would contain 9 quadrillion entries, which is too large to store on most systems. Instead, the `Output()` operation saves the result to a sparse matrix data structure. We discuss the sparse matrix data structure in more detail in the implementation section.

5. Achieving High-Performance

In the next few sections we present a simple analysis of the expected performance of the algorithm and introduce our model-driven protocol for implementing the algorithm. We also provide the specifications for the system used for the implementation.

5.1 Asymptotic Analysis

Asymptotic analysis is a useful technique for understanding how much of the theoretical peak performance an implementation can achieve. It is based on the assumption that memory reads and writes are where all performance is lost. By understanding the relationship between the number of reads and writes the algorithm requires, the general performance relative to the peak can be gauged. If the ratio of reads to writes is low or close to zero, once the processor has loaded the data, it then can process it without writing back to memory and thereby maintain a high rate of computation.

The dot plot algorithm requires $|\sum_{DNA}|n + m$ reads for $3mn$ operations. (The 3 is for the dot's (x, y, $value$) triple.) The number of writes is αmn, where $\alpha <= 1.0$ is the number of dots generated relative to the size of the full dot plot.

The general formula describing read-write/operations for the algorithm is:

$$\frac{|\Sigma_{DNA}|n + m + \alpha nm}{3nm}, \quad \alpha < 1$$

$|\sum_{DNA}|$ is very small - for DNA it is 4 - and this asymptotically goes to $\alpha / 3$, suggesting that we should be able to achieve near peak performance for large m and n and small α. The rate at which we approach this for any specific comparison will depend on α.

5.2 Modeling and Measuring Performance

To measure the target system's performance and set our expectations, we developed a set of models that contained the core operations of the algorithm. These were used initially to help understand the implementation parameters that impact design and also used during development to test implementation options. Due to the complexity of working with vector instructions, these simpler versions allowed us to experiment with different approaches and understand the impacts certain changes would have on the actual implementation.

The common inputs for all models are one or two data streams, represented by arrays of `unsigned chars` (`uchar`) and a `uchar*` result. The data streams simulate the sequences and the result pointer provides a storage location external to the model. Each model simulates one pass through the inner loop. Within each model, the number and types of instructions are varied according to the parameter we are testing.

All models were executed and timed together. Running all models this way helped ensure that the results were internally consistent and related to the same general system state. It also provided a feedback point for validating new models: if the values for known models were significantly different, the results may not be correct and the models were re-executed.

In addition to the performance models, we also maintained a stripped-down implementation of the DOTTER program that provided the same output as our program to use as the base performance measurement for the actual algorithm.

Each algorithm implementation had two versions that were used to gauge performance relative to the models. The *standard* version is the entire algorithm, as it would be used by an end-user. The *ideal* version contains the core of the algorithm but does not save the results. The ideal version provides an upper limit on the performance of the standard algorithm and is closer to the models.

5.3 Details

For development, testing, and benchmarking we used the complete genomes from the mitochondrial genome database, the chromosomes for yeast, and bacterial genomes from *E. Coli* and *Listeria*. All genomes were acquired from the NCBI Genbank database [4]. The collection of mitochondrial genomes provides small sequences [10-50kbp] with high levels of conservation between genomes. The high conservation rates tend to generate a high number of positive pairwise matches. The genome size and high number of positive matches enabled us to stress the algorithm on both the input and output streams. The yeast and bacterial genomes provided medium (230 – 1,100 kbp) and large (3-6 Mbp) length sequences, respectively, and were used to gauge the actual performance of the implementation.

The platform used to implement and test the algorithm was an Apple Dual 2 GHz PowerPC G5 with 3.5 GB DDR SDRAM running OS X 10.3.5 (Darwin Kernel Version 7.5.0). All code was compiled with g++ 3.3 (build 1620) from Xcode 1.5, with the -O3 –fast –altivec flags set. The VelocityEngine [3] using the Altivec instruction set was used for all vector operations. Version 1.31 of the Boost library [8] provided the thread and bind abstractions.

The time function used was `gettimeofday()` from `sys/time.h`. The resolution of `gettimeofday()` is 1 microsecond and has a calling time overhead of 70 nanoseconds and cost of about 78 flops. These values were determined experimentally on the target system.

Throughout this paper the term *op* refers to one base pair comparison. All performance measurements are in terms of this metric. Thus, 500 Mops is 500 million base pair comparisons per second.

6. Model-driven Implementation

The next few sections provide details of the models used to drive the development. The Altivec function calls are simply wrappers around assembly calls and the programmer handles all data loads manually. Because of this, the -altivec compiler flag turns off optimizations for vector code. This prevents the compiler from accidentally optimizing away an important instruction, but also places the burden of micro-optimization back on the developer. The model-driven approach to development helped us evaluate design decisions and understand the effects of different optimizations.

6.1 DOTTER Base

The DOTTER base model was created directly from the DOTTER source code. We removed all but the main dot plot loop for DNA/DNA comparisons to keep the comparison as fair as possible. We also used the same output data structures used in the vector implementation. The base implementation consistently yielded results of 130 Mops.

6.2 Data Stream Models

We developed a model to determine how to best access the data streams. The two approaches studied were incrementing all stream pointers and accessing the streams via indices. Because the core algorithm contains so few operations, the extra cost of either approach would have negative impacts on performance.

Three models were used to understand the effects of this decision. All models operated on one data stream to minimize the measurement effects from the additional stream operations. The first model, `VecAdd`, sums the values in the stream, indexing the array directly. `VecAddPointer` performs the same operation, but increments a pointer and dereferences it as needed to access the value. The final model, `VecAddEightPointer`, maintains eight pointers to the data stream that model the four window add and window subtract score vectors for each letter in $\sum_{DNA}$.

The results are listed in Table 1. Overall, `VecAddPointer` fared slightly better than `VecAdd`, suggesting that managing a pointer was more efficient for one data stream. However, as the number of pointers that were maintained increased, the performance decreased. The eight-pointer model was two-thirds the performance of the indexed version. Based on these results, an indexed array was used to access the score vectors and, because only one pointer is needed, an incrementing pointer was used for the lookup sequence.

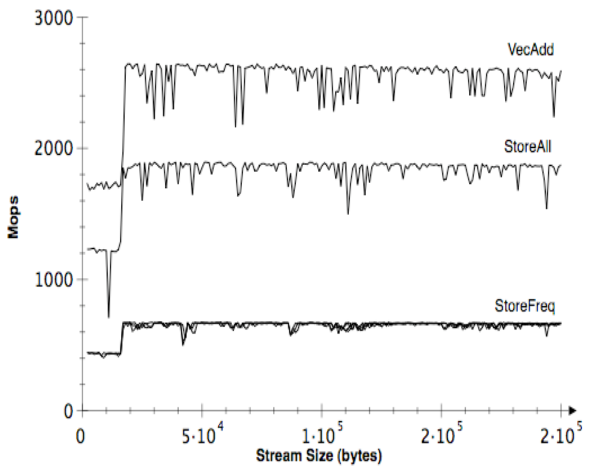

Figure 3 Vector Data Stream Performance. VecAdd shows rates for reading from two streams with one add and one subtract on the data. StoreAll is performs the same operations and also writes the results back to main memory. StoreFreq attempts to write the result back with a certain frequency determined by an integer mod operation.

6.3 Vector Performance Models

The base vector performance of the system was measured using the `VecAdd*` models, which read from two data streams, subtracting the current vector in the first string from the second and maintaining a running sum of the result. Three variations saved the results at different frequencies in order to model α. `VecAddTwo` provided the base implementation and stored no results. `VecAddTwoStoreAll` stored every result and `VecAddTwoStore` stored the result based on the current contents of the `newsum` vector and the frequency α. These models mimic the general structure of the vector dot plot algorithm and provide a rough upper limit on performance.

The results of the models are in (Figure 3). One measurement challenge evident in the results is that the attempt to model the frequency of stores introduced a dependency on the main processor for the `mod` operation. All values of α gave the exact same results. The branch stall caused by the mod operation prevented the frequency value from having an effect and instead simply provided a lower bound on performance. Of note is that the performance for storing *all* the results was significantly better when the dependency on branch from the main processor is removed. However, because of the potential for tera-

and peta-byte sized results for full genome comparisons, this is not yet feasible.

The results of the vector performance model provided an upper limit on performance of just over 2500 Mops and a lower limit of around 630 Mops for single CPU operation.

6.4 Sparse Matrix Format

Two different sparse matrix formats were used to store the results. The first attempt was based on the STL `std::vector` and consisted of a vector of vectors, the outer one representing the rows in the sparse matrix and the inner one containing a pair (*col*, *score*) for the column and score at that (*row*, *col*) position. The entire structure was kept in memory and serialized at the end of the run.

Table 2 Performance for different sparse matrix implementations

Matrix	Mops	Speedup
DOTTER base	130	N/A
std::vector	500	3.85
mmap	881	6.78

Initial comparisons between the ideal data-parallel implementation, the base version and the `std::vector` sparse matrix version were disappointing (Table 2). The potentially large speedup was almost entirely used up by the cost of maintaining the sparse matrix. Because the `std::vector` guarantees an optimal implementation for random data access, it would not be possible to further improve performance using it. Instead, we opted for a more direct representation and stored the triple (*row*, *col*, *score*) directly into a large integer array. To avoid potentially filling up main memory with results, they are written to a memory-mapped file via an accessesor function that managed the integer array memory and refreshed the mapped memory space as necessary. Using large mapped regions of 100 MB offset the extra cost of mapping memory.

6.5 Data Location

The location of the data plays an important role in the performance of the implementation. It is common practice to store large data sets in a shared location and also mount user directories from network drives, both for workstation use and in cluster environments. Because of this, program file I/O often requires network communication. We repeated the experiments used to select the sparse matrix format, but varied the location of the source and output files. The results are in Table 3.

Table 3 Effect of data location on performance

Matrix	NFS	Local	Local Speedup
std::vector	370	500	1.35x
mmap'd file	446	881	1.98x

Using the memory-mapped file, the speedup achieved by storing the sequences on results locally was almost 2x. The `std::vector` version also showed improved performance, but not as drastic as the memory-mapped version. Based on this result, all performance numbers were generated using local sequences and local result files.

6.5.1 Blocking and Prefetch

It is also worth noting that, other than the initial increase, the rates in Figure 3 do not improve as the data size grows. We ran this experiment with streams up to 500 MB and the same rate was maintained. Apple's high-performance development guide [3] suggests that the prefetch hardware in the G5 processor generally performs better than hand coded prefetch instructions. To verify this, we experimented with different prefetch strategies recommended for the G4 processor as well as various stream blocking algorithms. Every attempt to outperform the simple implementation failed to increase performance and in most cases caused a noticeable decrease.

6.6 Multi-processor Support

The embarrassingly parallel multi-processor implementation used the Boost.thread library [8] to distribute the *q* sequence across two processors. This roughly generated the expected 2x speedup.

6.7 Visualization

The dot plot technique of sequence comparison is inherently a visualization technique. However, generating images of large sequences presented a unique challenge. A typical display can directly show only about 1200 elements in each sequence, using one pixel for each dot. The pixel averaging techniques used by the DOTTER program support larger sequences with the main features of the matrix remaining visible, but some fine detail is lost. For sequences beyond 50 Kbp, there is no easy way to render the matrix without losing detail.

Our solution to this is simple and practical. Rather than rendering the matrices directly to the screen, we developed a tool that rendered the dots and diagonals to a PDF file. Dots were used for small comparisons and the diagonal lines were extracted and rendered for larger sequences to help manage the size of the PDF files. This allowed us to target output devices with various resolutions, including high-resolution displays and printers.

We rendered the images to 8.5x11 in or 120x120 in PDF files using a line size of 600 dpi. The larger size is the maximum size image we could print on a large format printer. For quick analysis, Adobe Acrobat provided the best rendering. Both Xpdf and Preview on OS X failed to render the 600 dpi lines when the whole image was displayed, leaving a blank screen. Acrobat rendered all the lines, regardless of the zoom level, and its antialiasing led to usable images, even at standard screen resolutions. In addition to rendering on traditional displays and paper, we also used Acrobat on a the IBM T221 3840x2400 pixel, 204 dpi display and a custom 2x4, 6400x2400 pixel tiled display wall. As the resolution and size increased, the amount of detail visible in the plots changed.

Paper printouts provided the most complete view of the comparisons, but for large comparisons, even the resolution of paper was too low to show all points. The high-resolution and large-format displays improved on the normal displays by showing finer details when the whole image was displayed. This made it easier to identify interesting areas to zoom-in on for further analysis.

7. Results and Discussion

The final results are listed in Table 4. The maximum performance for 2 CPUs peaked at about 1870 Mops and the single CPU performance reached 910 Mops, representing 7x and 14x speedups, respectively. The fastest ideal rates achieved were 4207 Mops on two CPUs and 2489 Mops on a single CPU, for speedups of 20.3x and 32.3, respectively. The single CPU ideal performance is about 6% less than that of the model. If this relation holds for the case were all results are stored to memory (model = 1890 Mops), a single CPU speed of 1777 Mops may be possible.

7.1 Target System Effects

While implementing serial versions of algorithms is a straightforward task, the implementation of the data-parallel dot plot shows that creating parallel versions of algorithms requires careful attention to the details of the implementation and the target system. As the results of the sparse matrix and NFS tests show, slight changes in the implementation and location of the data can have profound effects on the performance. Additionally, changes to the system environment can affect the performance.

Table 4 Final Results

Implementation	CPUs	Mops	Speedup
2 data stream model	1	2643	(20.3x)
Data-parallel, ideal	2	4207	(32.3x)
Data-parallel, ideal	1	2489	(19.1x)
Data-parallel, α=.04%	1	910	7.0x
Data-parallel α=.04%	2	1686	13.0x
Data parallel, large data	2	1868	14.4x

Our initial development was performed using g++ (build 1614) from Xcode 1.1. During the project, we upgraded to Xcode 1.5, which included "improvements for speed and –fast robustness" [3]. As expected, the vector code numbers remained the same. However, DOTTER's base performance improved significantly. Under the first compiler, it achieved a maximum rate of 90 Mops, 30% less than the rate from the new compiler. Using this rate, our peak speedup for one processor would be 10.1x, instead of 7.0x.

These results demonstrate an important point. To achieve maximum performance, the entire environment of the target system must be taken into account. Setting expectations up front with the models helped put our numbers in perspective and allowed us to understand the effects of the system. When actual numbers differed greatly from expectations, the models helped us probe the system for possible explanations. This helped us target areas outside of the core algorithm for optimization.

7.2 Problem Size

Table 5 lists the times to compare sequences of different lengths at the dual processor rate using commodity hardware. Assuming a linear speedup is achievable on a cluster of such machines, we also show extrapolated results for large sequences. Because of the lack of dependen-

Table 5 Actual and projected times needed to compare large sequences on commodity hardware. Single processor times are on actual values, cluster times are based on an ideal linear speedup for distributing the sequence across the cluster.

Sequence Size (m, n)	Time to compare at 1850 Mops
50,000 (Mitochondrial)	1 second
500,000 (Yeast Chromosomes)	2.2 minutes
5,000,000 (Bacteria)	3.8 hours
50,000,000 (Small human chromosomes)	15.6 days (2.5 hours on 1500 nodes)
500,000,000 (large human chromosomes)	1564 days (~1 day on 1500 nodes)
5,000,000,000 (full mammalian genomes)	104 days on 1500 nodes

cies between portions of the matrix, linear speedup is a reasonable assumption (although there may be the usual implementation challenges at larger scales). These numbers demonstrate that complete pairwise comparisons of large sequences are not only possible, but also practical for medium sized genomes. The difference in running times at the DOTTER base rate and the dual processor rate qualitatively changes the types of sequences that can be compared directly.

Currently, few techniques beyond visualizations exist for examining the complete pairwise comparison matrix. And techniques that use portions of the full matrix to help guide their execution, such as FASTA and LAGAN [5], are focused solely on producing linear alignments of sequences. Because of the challenges of visualizing large pairwise comparisons, the ability to produce the complete matrix for full genomes provides a new source of unfiltered data for developing comparative genomics tools.

8. Conclusion

We have presented a data-parallel and multiprocessor implementation of the dot plot algorithm and demonstrated some of the challenges in developing high-performance software to handle very large data sets. Our model-driven development style allowed us to take a rigorous approach to implementation and thoroughly explore the system parameters that affected it. We also demonstrated the feasibility of directly comparing large genomic sequences. The speedups attained by the data-parallel dot plot algorithm fundamentally change the nature of the questions that can be asked of genomes.

9. Acknowledgements

Jeremiah Willcock, from the Open Systems Lab at Indiana University, helped prototype an early version of the algorithm and provided feedback during the development process. Eric Wernert and Craig Jacobs from the Advanced Visualization Lab (AVL) at Indiana University provided visualization support for the IBM T221 display and the tiled display wall. This work was supported by a grant from the Lilly Endowment.

10. References

[1] S. F. Altschul, W. Gish, W. Miller, E. W. Myers and D. J. Lipman, *Basic local alignment search tool*, J Mol Biol, 215 (1990), pp. 403-10.

[2] (ACG) Apple Advanced Computation Group, Apple/Genentech BLAST, Apple, Cupertino, CA, 2002, http://www.apple.com/acg

[3] (ADC) Apple Developer's Connection, *Velocity Engine* and *Xcode*, from, *Apple Developer Connection*, Cupertino, CA, 2004. http://developer.apple.com/hardware/ve http://developer.apple.com/tools/xcode/

[4] D. A. Benson, I. Karsch-Mizrachi, D. J. Lipman, J. Ostell, B. A. Rapp and D. L. Wheeler, *GenBank*, Nucleic Acids Res, 28 (2000), pp. 15-8.

[5] M. Brudno, C. B. Do, G. M. Cooper, M. F. Kim, E. Davydov, E. D. Green, A. Sidow and S. Batzoglou, *LAGAN and Multi-LAGAN: efficient tools for large-scale multiple alignment of genomic DNA*, Genome Res, 13 (2003), pp. 721-31.

[6] O. Couronne, A. Poliakov, N. Bray, T. Ishkhanov, D. Ryaboy, E. Rubin, L. Pachter and I. Dubchak, *Strategies and tools for whole-genome alignments*, Genome Res, 13 (2003), pp. 73-80.

[7] C. Daggigian, *altivec-HMMER bechmark* [sic] *results from Erik Lindahl - 6x faster than dual Athlon*, in bioinformatics.org, ed., Biodevelopers mailing list, 2002. http://bioinformatics.org/pipermail/biodevelopers/2002-December/000111.html

[8] B. Dawes and D. Abrahams, *Boost*, www.boost.org, 2004.

[9] A. J. Gibbs and G. A. McIntyre, *The diagram, a method for comparing sequences. Its use with amino acid and nucleotide sequences*, Eur J Biochem, 16 (1970), pp. 1-11.

[10] Intel, *A-32 Intel® Architecture Software Developer's Manual, Volume 1: Basic Architecture*, *IA-32 Intel Architecture Software Developer's Manuals*, Intel, 2004. http://developer.intel.com/design/pentium4/manuals/index_new.htm

[11] J. V. Maizel, Jr. and R. P. Lenk, *Enhanced graphic matrix analysis of nucleic acid and protein sequences*, Proc Natl Acad Sci U S A, 78 (1981), pp. 7665-9.

[12] W. R. Pearson and D. J. Lipman, *Improved tools for biological sequence comparison*, Proc Natl Acad Sci U S A, 85 (1988), pp. 2444-8.

[13] T. Rognes and E. Seeberg, *Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors*, Bioinformatics, 16 (2000), pp. 699-706.

[14] T. F. Smith and M. S. Waterman, *Identification of common molecular subsequences*, J Mol Biol, 147 (1981), pp. 195-7.

[15] E. L. L. Sonnhammer and R. Durbin, *A Dot-Matrix Program with Dynamic Threshold Control Suited for Genomic DNA and Protein-Sequence Analysis*, Gene-Combis, 167 (1995), pp. 1-10.