How Perl Saved the Human Genome Project

by Lincoln Stein

Reprinted courtesy of the Perl Journal, http://www.tpj.com

Lincoln Stein's website is http://stein.cshl.org

DATE: Early February, 1996

LOCATION: Cambridge, England, in the conference room of the largest DNA sequencing center in Europe.

OCCASION: A high level meeting between the computer scientists of this center and the largest DNA sequencing center in the United States.

THE PROBLEM: Although the two centers use almost identical laboratory techniques, almost identical databases, and almost identical data analysis tools, they still can't interchange data or meaningfully compare results.

THE SOLUTION: Perl.

The human genome project was inaugurated at the beginning of the decade as an ambitious international effort to determine the complete DNA sequence of human beings and several experimental animals. The justification for this undertaking is both scientific and medical. By understanding the genetic makeup of an organism in excruciating detail, it is hoped that we will be better able to understand how organisms develop from single eggs into complex multicellular beings, how food is metabolized and transformed into the constituents of the body, how the nervous system assembles itself into a smoothly functioning ensemble. From the medical point of view, the wealth of knowledge that will come from knowing the complete DNA sequence will greatly accelerate the process of finding the causes of (and potential cures for) human diseases.

Six years after its birth, the genome project is ahead of schedule. Detailed maps of the human and all the experimental animals have been completed (mapping out the DNA using a series of landmarks is an obligatory first step before determining the complete DNA sequence). The sequence of the smallest model organism, yeast, is nearly completed, and the sequence of the next smallest, a tiny soil-dwelling worm, isn't far behind. Large scale sequencing efforts for human DNA started at several centers a number of months ago and will be in full swing within the year.

The scale of the human DNA sequencing project is enough to send your average Unix system administrator running for cover. From the information-handling point of view, DNA is a very long string consisting of the four letters G, A, T and C (the letters are

abbreviations for the four chemical units that form the "rungs" of the DNA double helix ladder). The goal of the project is to determine the order of letters in the string. The size of the string is impressive but not particularly mind-boggling: 3 x 10^9 letters long, or some 3 gigabytes of storage space if you use 1 byte to store each letter and don't use any compression techniques.

Three gigabytes is substantial but certainly manageable by today's standards. Unfortunately, this is just the storage space required to store the finished data. The storage requirements for the experimental data needed to determine this sequence is far more vast. The essential problem is that DNA sequencing technology is currently limited to reading stretches of at most 500 contiguous letters. In order to determine sequences longer than that, the DNA must be sequenced as small overlapping fragments called "reads" and the jigsaw puzzle reassembled by algorithms that look for areas where the sequences match. Because the DNA sequence is nonrandom (similar but not-entirely-identical motifs appear many times throughout the genome), and because DNA sequencing technology is noisy and error-prone, one ends up having to sequence each region of DNA five to 10 times in order to reliably assemble the reads into the true sequence. This increases the amount of data to manage by an order of magnitude. On top of this is all the associated information that goes along with laboratory work: who performed the experiment, when it was performed, the section of the genome was sequenced, the identity and version of the software used to assemble the sequence, any comments someone wants to attach to the experiment, and so forth. In addition, one generally wants to store the raw output from the machine that performs the sequencing itself. Each 500 letters of sequence generates a data file that's 20-30 kilobytes long!

That's not the whole of it. It's not enough just to determine the sequence of the DNA. Within the sequence are functional areas scattered among long stretches of nonfunctional areas. There are genes, control regions, structural regions, and even a few viruses that got entangled in human DNA long ago and persist as fossilized remnants. Because the genes and control regions are responsible for health and disease, one wants to identify and mark them as the DNA sequence is assembled. This type of annotation generates yet more data that needs to be tracked and stored.

Altogether, people estimate that some 1 to 10 terabytes of information will need to be stored in order to see the human genome project to its conclusion.

So what's Perl got to do with it? From the beginning researchers realized that informatics would have to play a large role in the genome project. An informatics core formed an integral part of every genome center that was created. The mission of these cores was two-fold: to provide computer support and databasing services for their affiliated laboratories, and to develop data analysis and management software for use by the genome community as a whole.

It's fair to say that the initial results of the informatics groups efforts were mixed. Things were slightly better on the laboratory management side of the coin. Some groups attempted to build large monolithic systems on top of complex relational databases; they were thwarted time and again by the highly dynamic nature of biological research. By the time a system that could deal with the ins and outs of a complex laboratory protocol had been designed, implemented and debugged, the protocol had been superseded by new technology and the software engineers had to go back to the drawing board.

Most groups, however, learned to build modular, loosely-coupled systems whose parts could be swapped in and out without retooling the whole system. In my group, for example, we discovered that many data analysis tasks involve a sequence of semi-independent steps. Consider the steps that one may want to perform on a bit of DNA that has just been sequenced (Figure 1). First there's a basic quality check on the sequence: is it long enough? Are the number of ambiguous letters below the maximum limit? Then there's the "vector check". For technical reasons, the human DNA must be passed through a bacterium before it can be sequenced (this is the process of "cloning"). Not infrequently, the human DNA gets lost somewhere in the process and the sequence that's read consists entirely of the bacterial vector. The vector check ensures that only human DNA gets into the database. Next there's a check for repetitive sequences. Human DNA is full of repetitive elements that make fit! ting the sequencing jigsaw puzzle together challenging. The repetitive sequence check tries to match the new sequence against a library of known repetitive elements. A penultimate step is to attempt to match the new sequence against other sequences in a large community database of DNA sequences. Often a match at this point will provide a clue to the function of the new DNA sequence. After performing all these checks, the sequence along with the information that's been gathered about it along the way is loaded into the local laboratory database.

The process of passing a DNA sequence through these independent analytic steps looks kind of like a pipeline, and it didn't take us long to realize that a Unix pipe could handle the job. We developed a simple Perl-based data exchange format called "boulderio" that allowed loosely coupled programs to add information to a pipe-based I/O stream. Boulderio is based on tag/value pairs. A Perl module makes it easy for programs to reach into the input stream, pull out only the tags they're interested in, do something with them, and drop new tags into output the stream. Any tags that the program isn't interested in are just passed through to standard output so that other programs in the pipeline can get to them.

Using this type of scheme, the process of analyzing a new DNA sequence looks something like this (this is not exactly the set of scripts that we use, but it's close enough):

```
name_sequence.pl < new.dna |

quality_check.pl |

vector_check.pl |

find_repeats.pl |

search_big_database.pl |

load_lab_database.pl
```

A file containing the new DNA sequence is processed by a perl script named "name_sequence.pl", whose only job is to give the sequence a new unique name and to put it into boulder format. Its output looks like this:

```
NAME=L26P93.2

SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC.....
```

The output from name_sequence.pl is next passed to the quality checking program, which looks for the SEQUENCE tag, runs the quality checking algorithm, and writes its conclusion to the data stream. The data stream now looks like this:

```
NAME=L26P93.2

SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC.....

QUALITY_CHECK=OK
```

Now the data stream enters the vector checker. It pulls the SEQUENCE tag out of the stream and runs the vector checking algorithm. The data stream now looks like this:

```
NAME=L26P93.2

SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC.....

QUALITY_CHECK=OK

VECTOR_CHECK=OK
```

```
VECTOR_START=10

VECTOR_LENGTH=300
```

This continues down the pipeline, until at last the "load_lab_database.pl" script collates all the data collected, makes some final conclusions about whether the sequence is suitable for further use, and enters all the results into the laboratory database.

One of the nice features of the boulderio format is that multiple sequence records can be processed sequentially in the same Unix pipeline. An "=" sign marks the end of one record and the beginning of the next:

```
NAME=L26P93.2

SEQUENCE=GATTTCAGAGTCCCAGATTTCCCCCAGGGGGTTTCCAGAGAGCCC.....

=

NAME=L26P93.3

SEQUENCE=CCCCTAGAGAGAGAGAGCCGAGTTCAAAGTCAAAACCCATTCTCTCTCCTC...

=
```

There's also a way to create subrecords within records, allowing for structured data types.

Here's an example of a script that processes boulderio format. It uses an object-oriented style, in which records are pulled out of the input stream, modified, and dropped back in:

```
use Boulder::Stream;

$stream = new Boulder::Stream;

while ($record = $stream->read_record('NAME','SEQUENCE')) {

$name = $record->get('NAME');

$sequence = $record->get('SEQUENCE');

...[continue processing]...

$record->add(QUALITY_CHECK=>"OK");
```

```
$stream->write_record($record);

}
```

(If you're interested, more information about the boulderio format and the perl libraries to manipulate it can be found at http://stein.cshl.org/software/boulder/).

The interesting thing is that multiple informatics groups independently converged on solutions that were similar to the boulderio idea. For example, several groups involved in the worm sequencing project began using a data exchange format called ".ace". Although this format was initially designed as the data dump and reload format for the ACE database (a database specialized for biological data), it happens to use a tag/value format that's very similar to boulderio. Soon .ace files were being processed by Perl script pipelines and loaded into the ACE database at the very last step.

Perl found uses in other aspects of laboratory management. For example, many centers, including my own, use Web based interfaces for displaying the status of projects and allowing researchers to take actions. Perl scripts are the perfect engine for Web CGI scripts. Similarly, Perl scripts run e-mail database query servers, supervise cron jobs, prepare nightly reports summarizing laboratory activity, create instruction files to control robots, and handle almost every other information management task that a busy genome center needs.

So as far as laboratory management went, the informatics cores were reasonably successful. As far as development and distributing generally useful, however, things were not so rosy.

The problem will be familiar to anyone who has worked in a large, loosely organized software project. Despite best intentions, the project begins to drift. Programmers go off to work on ideas that interest them, modules that need to interface with one another are designed independently, and the same problems get solved several times in different, mutually incompatible ways. When the time comes to put all the parts together, nothing works.

This is what happened in the genome project. Despite the fact that everyone was working on the same problems, no two groups took exactly the same approach. Programs to solve a given problem were written and rewritten multiple times. While a given piece of software wasn't guaranteed to work better than its counterpart developed elsewhere, you could always count on it to sport its own idiosyncratic user interface and data format. A typical example is the central algorithm that assembles thousands of short DNA reads into an ordered set of overlaps. At last count there were at least six different programs in widespread use, and no two of them use the same data input or output formats.

This lack of interchangeability presents terrible dilemma for the genome centers. Without interchangeability, an informatics group is locked into using the software that it developed in-house. If another genome center has come up with a better software tool to attack the same problem, a tremendous effort is required by the first center to retool their system in order to use that tool.

The long range solution to this problem is to come up with uniform data interchange standards that genome software must adhere to. This would allow common modules to be swapped in and out easily. However, standards require time to agree on, and while the various groups are involved in discussion and negotiation, there is still an urgent need to adapt existing software to the immediate needs of the genome centers.

Here is where Perl again came to the rescue. The Cambridge summit meeting that introduced this article was called in part to deal with the data interchange problem. Despite the fact that the two groups involved were close collaborators and superficially seemed to be using the same tools to solve the same problems, on closer inspection nothing they were doing was exactly the same.

The main software components in a DNA sequencing projects are:

- a trace editor to analyze, display and the short DNA read chromatograms from DNA sequencing machines.
- a read assembler, to find overlaps between the reads and assemble them together into long contiguous sections.
- an assembly editor, to view the assemblies and make changes in places where the assembler went wrong.
- a database to keep track of it all.

Over the course of a few years, the two groups had developed suites of software that worked well in their hands. Following the familiar genome center model, some of the components were developed in-house while others were imported from outside. As shown in Figure 2, Perl was used for the glue to make these pieces of software fit together. Between each pair of interacting modules were one or more Perl scripts responsible for massaging the output of one module into the expected input for another.

When the time came to interchange data, however, the two groups hit a snag. Between them they were now using two trace editors, three assemblers, two assembly editors and (thankfully) one database. If two Perl scripts were required for each pair of components (one for each direction), one would need as many as 62 different scripts to handle all the possible interconversion tasks. Every time the input or ouput format of one of these modules changed, 14 scripts might need to be examined and fixed.

The solution that was worked out during this meeting is the obvious one shown in Figure 3. The two groups decided to adopt a common data exchange format known as CAF (an

acronym whose exact meaning was forgotten during the course of the meeting). CAF would contain a superset of the data that each of the analysis and editing tools needed. For each module, two Perl scripts would be responsible for converting from CAF into whatever format Module A expects ("CAF2ModuleA") and converting Module A's output back into CAF ("ModuleA2CAF"). This simplified the programming and maintenance task considerably. Now there were only 16 Perl scripts to write; when one module changed, only two scripts would need to be examined.

This episode is not unique. Perl has been the solution of choice for genome centers whenever they need to exchange data, or to retrofit one center's software module to work with another center's system.

So Perl has become the software mainstay for computation within genome centers as well as the glue that binds them together. Although genome informatics groups are constantly tinkering with other "high level" languages such as Python, Tcl and recently Java, nothing comes close to Perl's popularity. How has Perl achieved this remarkable position?

I think several factors are responsible:

1. Perl is remarkably good for slicing, dicing, twisting, wringing, smoothing, summarizing and otherwise mangling text. Although the biological sciences do involve a good deal of numeric analysis now, most of the primary data is still text: clone names, annotations, comments, bibliographic references. Even DNA sequences are textlike. Interconverting incompatible data formats is a matter of text mangling combined with some creative guesswork. Perl's powerful regular expression matching and string manipulation operators simplify this job in a way that isn't equalled by any other modern language.

2. Perl is forgiving. Biological data is often incomplete, fields can be missing, or a field that is expected to be present once occurs several times (because, for example, an experiment was run in duplicate), or the data was entered by hand and doesn't quite fit the expected format. Perl doesn't particularly mind if a value is empty or contains odd characters. Regular expressions can be written to pick up and correct a variety of common errors in data entry. Of course this flexibility can be also be a curse. I talk more about the problems with Perl below.

3. Perl is component-oriented. Perl encourages people to write their software in small modules, either using Perl library modules or with the classic Unix tool-oriented approach. External programs can easily be incorporated into a Perl script using a pipe, system call or socket. The dynamic loader introduced with Perl5 allows people to extend the Perl language with C routines or to make entire compiled libraries available for the Perl interpreter. An effort is currently under way to gather all the world's collected wisdom about biological data into a set of modules called "bioPerl" (discussed at length in an article to be published later in the Perl Journal).

4. Perl is easy to write and fast to develop in. The interpreter doesn't require you to declare all your function prototypes and data types in advance, new variables spring into existence as needed, calls to undefined functions only cause an error when the function is needed. The debugger works well with Emacs and allows a comfortable interactive style of development.

5. Perl is a good prototyping language. Because Perl is quick and dirty, it often makes sense to prototype new algorithms in Perl before moving them to a fast compiled language. Sometimes it turns out that Perl is fast enough so that of the algorithm doesn't have to be ported; more frequently one can write a small core of the algorithm in C, compile it as a dynamically loaded module or external executable, and leave the rest of the application in Perl (for an example of a complex genome mapping application implemented in this way, see http://waldo.wi.mit.edu/ftp/distribution/software/rhmapper/).

6. Perl is a good language for Web CGI scripting, and is growing in importance as more labs turn to the Web for publishing their data.

My experience in using Perl in a genome center environment has been extremely favorable overall. However I find that Perl has its problems too. Its relaxed programming style leads to many errors that more uptight languages would catch. For example, Perl lets you use a variable before its been assigned to, a useful feature when that's what you intend but a disaster when you've simply mistyped a variable name. Similarly, it's easy to forget to declare make a variable used in a subroutine local, inadvertently modifying a global variable.

If one uses the **-w** switch religiously and turn on the **"use strict vars"** pragma, these Perl will catch these problems and others. However there are more subtle gotchas in the language that are not so easy to fix. A major one is Perl's lack of type checking. Strings, floats and integers all interchange easily. While this greatly speeds up development, it can cause major headaches. Consider a typical genome center Perl script that's responsible for recording the information of short named subsequences within a larger DNA sequence. When the script was written, the data format was expected to consist of tab-delimited fields: a string followed by two integers representing the name, starting position and length of a DNA subsequence within a larger sequence. An easy way to parse this would to split() into an list like this:

```
($name,$start_position,$length) = split("\t");
```

Later on in this script some arithmetic is performed with the two integer values and the result written to a database or to standard output for further processing.

Then one day the input file's format changes without warning. Someone bumps the field count up by one by sticking a comment field between the name and the first integer. Now the unknowing script assigns a string to a variable that's expected to be numeric and silently discards the last field on the line. Rather than crashing or returning an error code, the script merrily performs integer arithmetic on a string, assuming a value of zero for the string (unless it happens to start with a digit). Although the calculation is meaningless, the output may look perfectly good, and the error may not be caught until some point well downstream in the processing.

The final Perl deficiency has been a way to create graphical user interfaces. Although Unix True Believers know that anything worth doing can be done on the command line, most end-users don't agree. Windows, menus and bouncing icons have become de rigueur for programs intended for use by mere mortals.

Until recently, GUI development in Perl was awkward to impossible. However the work of Nick Ing-Simmons and associates on perlTK (pTK) has made Perl-driven GUIs possible on X-windows systems. My associates and I have written several pTK-based applications for internal use at the MIT genome center, and it's been a satisfying experience overall. Other genome centers make much more extensive use of pTK, and in some places its become a mainstay of production.

Unfortunately, I'm sad to confess that a few months ago when I needed to put a graphical front end on a C++ image analysis program I'd written, I turned to the standard Tcl/Tk library rather than to pTK. I made this choice because I intended the application for widespread distribution. I find pTK still too unstable for export: new releases of pTK discover lurking bugs in Perl, and vice-versa. Further, I find that even seasoned system administrators run into glitches when compiling and installing Perl modules, and I worried that users would hit some sort of road block while installing either pTK or the modules needed to support my application, and owuld give up. In contrast, many systems have the Tcl/Tk libraries preinstalled; if they don't, installation is quick and painless.

In short, when the genome project was foundering in a sea of incompatible data formats, rapidly-changing techniques, and monolithic data analysis programs that were already antiquated on the day of their release, Perl saved the day. Although it's not perfect, Perl seems to fill the needs of the genome centers remarkably well, and is usually the first tool we turn to when we have a problem to solve.