Accelerating HMMer searches on Opteron processors with minimally invasive recoding

Joseph Landman, Scalable Informatics LLC (landman@scalableinformatics.com)
Joydeep Ray, Advanced Micro Devices (joydeep.ray@amd.com)
J.P. Walters, Wayne State University (jwalters@wayne.edu)

Abstract

HMMer is a widely used tool for protein sequence homology detection, as well as functional annotation of homologous protein sequences, and protein family classification. The HMMer program is based upon a Viterbi algorithm coded in C, and is quite time consuming. Significant efforts have been undertaken to accelerate this program using custom special purpose hardware, as well as more recent attempts to leverage commodity special purpose hardware. This work will report on several minimally invasive code refactoring efforts independently undertaken by the authors, and their significant performance impact on wall clock execution time of the entire program for various test cases.

Introduction

Protein sequence analysis tools to predict homology, structure and function of particular peptide sequences exist in abundance. One of the most commonly used tools is the profile hidden Markov model algorithm developed by Eddy [Eddy98] and coworkers [Durbin98]. These tools allow scientists to construct mathematical models (Hidden Markov Models or HMM) of a set of aligned protein sequences with known similar function and homology, which is then applicable to a large corpus (database) of proteins. The tools provide the ability to generate a log-odds score as to whether or not the protein belongs to the same family as the proteins which generated the HMM, or to a set of random unrelated sequences.

Due to the complexity of the calculation, and the possibility to apply many HMM's to a single sequence (pfam search), these calculations require significant numbers of processing cycles. Efforts to accelerate these searches have resulted in several platform and hardware specific variants including an Altivec port by Lindahl [Lindahl05], a GPU port of hmmsearch by Stanford team [Horn05]. More recently one of the authors has completed an MPI and SSE2 modification of HMMer [Walters06].

These efforts span a range between minimal source code changes with some impact upon performance, to recasting the core algorithms in terms of a different computing technology and thus fundamentally altering the calculation. Each approach has specific benefits and costs. The approaches as reported in this paper were developed independently at approximately the same time, though the MPI port preceded the other work by several months.

HMMer in a nutshell

HMMer operations rely upon accurate construction of an HMM representation of a multiple sequence alignment (MSA) of homologous protein sequences. This HMM may then be applied to a corpus (database) of other protein sequences for homology determination, or grouped together to form part of a protein family set of HMMs that are used to test whether a particular protein sequence is related to the consensus model, and annotate potential functions within the query protein from what is known about the function of the aligned sequence from the HMM (homology transfer and functional inference). These functions in HMMer are based upon the profile HMM [Eddy98] architecture. The profile HMM architecture is constructed using the plan-7 model.

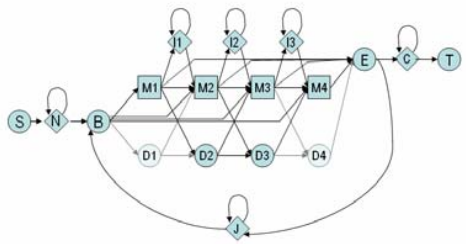


Figure 1. Plan 7 HMM model architecture

This architecture encodes insert, deletion, and match states all relative to a consensus sequence model. The plan-7 architecture is a Viterbi algorithm [Viterbi67] and the code implementing the plan-7 architecture is constructed as such. Viterbi algorithms involve state vector

Proceedings of the 20th International Conference on Advanced Information Networking and Applications (AINA'06)
1550-445X/06 $20.00 © 2006 IEEE

initialization, comparison operations to compute the most probable path to the subsequent state, and thus at the end of the algorithm, the most probable/maximum likelihood (Viterbi) path through the state trellis. The application of the previously constructed HMM to the protein sequence data will generate the aforementioned log-odds score and an optimal alignment represented by the Viterbi path.

This architecture allows for detection of remote homology in a sequence with significant numbers of insertions and deletions, as well as repeats and other phenomenon. Some variance between profile HMM implementations are seen using different architectures [Madera02]. The architecture will generally perform well at its homolog detection function for a well-constructed MSA with highly conserved regions.

Analysis / optimization of the hotspots:

The work described below was performed independently at approximately the same time by authors JL and JR, who learned of the others work after the fact. The approaches and changes to the HMMer source that were created were strikingly similar to the point that the authors decided that simultaneous joint reporting of the effort was indicated. Author JL was familiar with author JPW's work on SSE2 and MPI, and discussed this work.

The benchmark test cases

Author JL used the Bioinformatics Benchmark System v3 (BBSv3) [Landman03] test cases for HMMer. These test cases were based upon a benchmark suggested by Professor Eddy. They utilize identical code paths across the application suite. They are based upon generation of random databases and random query strings to test a generated HMM. The globin HMM was generated as per the example in the HMMer user guide. Author JR initially used HMMer test cases that are representative of real-world usage of this program. Later, for uniformity, he tested his improvements with the BBSv3 test cases for HMMer.

Author JL built and profiled the HMMer code with threading disabled. The test cases were executed, and the resulting profile examined. The benchmarks indicated approximately 97% of the execution time on the BBSv3 test cases was spent in the P7Viterbi routine in the HMMer source file fast_algorithms.c. Indeed, careful inspection of this file indicates the source code contribution from Lindahl, which represents a significant rewrite of the core Viterbi algorithm to map it onto the Altivec instruction set architecture (with support from the gcc compiler). Unfortunately, gcc is unable to emit SSE2 equivalents for the language extensions placed into the Altivec version, which subsequently renders the Altivec specific code non-portable without rewriting it.

Since the P7Viterbi routine consumes the most time in this run, an examination of how it was constructed, where it spent most of its time, and what relatively small changes could be effected that introduced large performance effects (which would also be portable across all architectures) was undertaken.

Author JPW reports elsewhere [Walters05] on efforts to recast the P7Viterbi routine in terms of SSE2 code in addition to leveraging MPI for a hierarchical acceleration method. Both the authors JL and JR were interested in a minimally invasive a set of changes as possible to maximize performance. There are fundamental limitations with this minimally invasive approach, in that the underlying computational hardware will not likely be utilized as efficiently as a retargeting/rewriting of the relevant code. The changes will be portable though, and should positively impact all users of the code.

Authors JL and JR diverged slightly in approaches to the problem of improving P7Viterbi performance. As indicated before, the similarity of the approaches is remarkable, as are the code patches and performance characteristics.

Author JL's approach

Extending the profiling of the code down to the line level with long test cases indicated that the conditionals and the loop in the P7Viterbi routine were consuming approximately 30% and 60% respectively of the execution time of this routine. Carefully examining the loop, several issues were immediately obvious: First, the variable sc was superfluous, and forced a memoization of the value of an intermediate calculation. Without a complete aliasing analysis on the part of the compiler, there would be little opportunity for the optimizer to remove the variable, and leave the intermediate results in a register.

Second, the use of variable sc resulted in creating artificial dependencies between different sections of the loop and between iterations of the loop. The former would prevent the optimizer from moving statements around to reduce

memory traffic and better utilize processor resources. The latter would impede automatic unrolling.

Finally, the conditional within the loop is always executed except for the last iteration. This implies that this loop can be refactored into two loops, one with k incrementing from 1 to M-1 over the main loop block (MLB) with no conditional needed to execute the conditional block (CB), and one main loop block with k == M without the conditional block. That is we alter the structure of the loop from figure 2 to that of figure 3.

```
for (k = 1; k <= M; k++) {
    /* main loop block:  MLB as a function of k */
            …
     if (k < M) {
       /* conditional block: CB */
             …
     }
}
```

Figure 2. Representation of expensive loop

```
for (k = 1; k < M; k++) {
    /* main loop block:  MLB as a function of k */
            …
     /* conditional block: CB */
            …
}
k=M;
/* main loop block:  MLB at k=M */
…
```

Figure 3. Refactored expensive loop

Removing the conditional from the loop lets the compiler generate better code for the loop as long as the artificial loop iteration dependency was broken by removing the memoization of sc and using a register temporary. After making these changes, the HMMer regression tests included with the code distribution were rerun, and the benchmark output was inspected to insure correctness of the calculations.

This set of changes resulted in a binary approximately 1.8x faster than the binary built from the original source tree. An inter-loop dependency still exists with the use of the sc1 variable. However, the only time memory traffic will be observed will be when the assignment conditional is true, which should allow the cached value of sc to be used without requiring repeated memoization where it is not required. Additional work was performed on the function to remove the sc variable from other conditionals so as to avoid the artificial dependencies.

Combining these changes with the preceding changes yielded a binary approximately 1.96x faster than the Opteron baseline binary provided by the HMMer download site. Subsequent tests on end user cases have yielded a range from 1.6x to 2.5x baseline performance, depending in part upon which database and how the HMM was constructed for the tests.

Author JR's approach

Author JR started with profiling the code with the linux profiler *oprofile*. The profiler indicated that 96% of the runtime is spent in the inner loop of P7Viterbi function, as found by author JL in Figure 1.

JR then looked at the machine instructions generated by the *gcc* compiler for this loop, and observed that the potential chance of aliasing between arrays that are written (mc[k], dc[k] and ic[k]) with arrays that are read (ip[k-1], tpim[k-1], dpp[k-1] etc) is causing the compiler to generate sub-optimal code. mc[k], dc[k] and ic[k] values are not kept in register in the loop body, they are saved to memory each time they are computed, before other array variables are read to evaluate the next conditional. However, these arrays are not meant to alias in the program, but it is hard for a compiler to determine that automatically.

Secondly, the gcc compiler was not generating the conditional move instructions *(cmov),* available in the x86 and x86-64 instruction set, for the small conditionals in the loop body. Use of *cmov* instructions for branches that are hard to predict is usually a big win because it avoids the mis-prediction penalty when the branches are incorrectly predicted. Since the conditionals in the loop body write to a memory location (e.g. *mc[k] = sc;*), the compiler must determine that it is safe to write to that memory location *unconditionally*, in order to convert that conditional to its *cmov* form. Apparently, the *gcc* compiler was not able to do this automatically.

Author JR re-factored the source to fix the two above-mentioned problems. The resulting code was very similar to that of author JL: use of temporaries (local variables) to store the intermediate computations (Figure 8). This change eliminated the important potential false aliasings and also writing to memory inside the conditionals. With these small changes, the machine code generated by *gcc* uses *cmov* instructions for *all* the conditionals in the loop body.

```
for (k = 1; k < M; k++) {
  sc1 = mpp[k-1]   + tpmm[k-1];
  if ((ip[k-1]+tpim[k-1]) > sc1)
          sc1 = ip[k-1]+tpim[k-1];
  if ((dpp[k-1]+tpdm[k-1])> sc1)
          sc1 = dpp[k-1]+tpdm[k-1] ;
  if (( xmb  + bp[k])     > sc1)
          sc1 = xmb+bp[k] ;
```

Figure 8. Main loop with conditionals

Another problem observed by JR is that GCC's register allocation for the loop body is not optimal. The number of integer registers available in x86 and x86-64 architecture is not adequate for holding all the array bases, constants and temporaries referenced in the inner loop. JR split the loop into two sub-loops to help register allocation.

```
for (k = 1; k <= M; k++) {
   /* main loop block:  MLB as a function of k */
   sc1 = mpp[k-1]   + tpmm[k-1];
   if ((ip[k-1]+tpim[k-1]) > sc1)
           sc1 = ip[k-1]+tpim[k-1];
   if ((dpp[k-1]+tpdm[k-1])> sc1)
            sc1 = dpp[k-1]+tpdm[k-1] ;
   if (( xmb  + bp[k])     > sc1)
           sc1 = xmb+bp[k] ;
   sc1 += ms[k];
   if (sc1 < -INFTY) sc1 = -INFTY;
   mc[k] = sc1;
   sc2 = dc[k-1] + tpdd[k-1];
   if ((mc[k-1]+tpmd[k-1])>sc2)
           sc2 = mc[k-1]+tpmd[k-1] ;
   if (sc2 < -INFTY) sc2 = -INFTY;
   dc[k] = sc2;
}

for (k = 1; k < M; k++) {
   sc3 = mpp[k] + tpmi[k];
   if ((ip[k] + tpii[k]) > sc3) sc3 = ip[k]+tpii[k] ;
   sc3 += is[k];
   if (sc3 < -INFTY) sc3 = -INFTY;
   ic[k] = sc3;
  }
```

Figure 9. Time expensive loop in P7Viterbi after refactoring

Combining these changes sped up HMMer upto **1.6X** compared to the stock HMMer source code, both compiled with "-O3". The changes were verified using the regression tests supplied with HMMer. The compiler used is *gcc* 3.3.3 for x86-64.

Author JPW's approach and summary of previously reported work

As discussed in [Walters05], an MPI + SSE2 based implementation has been independently developed to effectively utilize a computational cluster in order to accelerate HMMer searches on x86 based hardware. In this case, the changes required to parallelize the HMMer algorithm were much more extensive, than those we propose here. The tradeoff is that in our implementation we seek a smaller speedup but with a minimum of programming overhead.

In this case of cluster based implementation, the work was split into two complementary components: an SSE2 implementation and an MPI implementation. The SSE2 implementation sought to vectorize the critical P7Viterbi() function and a modest speedup was observed. This was due largely to the Pentium 4's rather anemic SIMD hardware/instruction set.

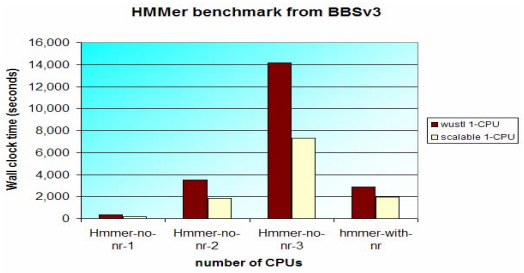
The MPI implementation utilized a double-buffering scheme to ensure that worker nodes spent a minimal amount of time waiting for new data to arrive. The strategy utilized a master node whose job was to simply ensure that each worker had data, and to collect the workers' output. The master performed a minimum amount of post-processing on the data.

In this case, the changes were made entirely external to the P7Viterbi() function, allowing the SSE2 code to be included or excluded according to a user's desires. The MPI solution proved quite scalable through tests of 16 workers.

Results

The binary generated by JL's method was compared to the standard downloadable Opteron binary, running the BBSv3 tests. No special efforts were undertaken to make the machine quiescent prior to the run, other than to ascertain whether or not another user was running jobs.

We note that the results shown below demonstrate a greater speedup than the SSE2 implementation described above. Given that the amount of recoding necessary to achieve this speedup was far less than that of the SSE2 implementation, we feel that our approach is reasonable for SMP based machines. Single CPU results are summarized in the following graph.



1550-445X/06 $20.00 © 2006 IEEE

Figure 10. Single CPU performance comparison between the canonical WU distributed binary, and the Scalable Informatics generated binary.

Table 1. measured results for the Washington University binary and the Scalable Informatics generated binary.

Test	Binary version	NCPU	Wall clock (s)	Perf ratio
Hmmer-no-nr-1	scalable	1	174.2	1.96
Hmmer-no-nr-1	scalable	2	88.4	3.86
Hmmer-no-nr-1	wustl	1	341.7	1.00
Hmmer-no-nr-1	wustl	2	173.5	1.97
Hmmer-no-nr-2	scalable	1	1813.4	1.94
Hmmer-no-nr-2	scalable	2	948.4	3.71
Hmmer-no-nr-2	wustl	1	3522.2	1.00
Hmmer-no-nr-2	wustl	2	1968.5	1.79
Hmmer-no-nr-3	scalable	1	7292.8	1.94
Hmmer-no-nr-3	scalable	2	3765.3	3.76
Hmmer-no-nr-3	wustl	1	14143.1	1.00
Hmmer-no-nr-3	wustl	2	7065.4	2.00
hmmer-with-nr	scalable	1	1969.7	1.47
hmmer-with-nr	scalable	2	661.5	4.37
hmmer-with-nr	wustl	1	2888.7	1.00
hmmer-with-nr	wustl	2	1096.6	2.63

Future work:

We have shown that modest improvements performance improvements can be made to the HMMer suite with a minimum of re-programming. Further improvements will likely involve more involved programming. To that end we suggest integrating the improvements described in section 3 with that described in [Walters05].

Specifically, we suggest that the improvements described in this paper may complement (or perhaps replace) the small improvements provided by the SSE2 implementation. This could simply be plugged into the existing MPI code for parallelized/clustered HMMer searches.

Acknowledgements:

One of the authors (JL) is indebted to the many helpful discussions with Professor Vipin Chaudhary at Wayne State University, and Professor Sean Eddy of Washington University St. Louis. Author JR is indebted to Dwarak Rajagopal, Evandro Menezes and Tony Linthicum (all AMD employees) for help with the GCC compiler's code generation.

Bibliography

[Durbin98] Durbin, R., Eddy, S. R., Krogh, A., and Mitchison, G. J. (1998). *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, Cambridge UK.

[Eddy98] Eddy, S. R. (1998). Profile hidden Markov models. *Bioinformatics*, 14:755–763.

[Horn05] Horn, D.R. , Houston, M. , Hanrahan, P. (2005) *ClawHMMER: A Streaming HMMer-Search Implementation,* SC05 (http://graphics.stanford.edu/papers/clawhmmer/hmmer.pdf)

[Landman03] Landman, J.I. (2003) unpublished work, code and test cases available at http://www.scalableinformatics.com/bbs

[Lindahl05] Lindahl, E., 2005. Altivec HMMer, version 2. http://lindahl.sbc.su.se/software/altivec/altivec-hmmer-version-2.html

[Madera02] Madera, M., Gough, J.,(2002) *A comparison of profile hidden Markov model procedures for remote homology detection,* Nucleic Acids Research, 2002, Vol. 30, No. 19 4321-4328, Oxford University Press

[Viterbi67] Viterbi, A.J., "*Error bounds for convolutional codes and an asymptotically optimum decoding algorithm*," IEEE Trans. Inform. Theory, vol. IT–13, pp. 260–269, April 1967.

[Walters06] Walters, J.P. , Qudah, B., and Chaudhary, V., (2005) *Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors*, To Appear in AINA 2006, Vienna, Austria.