Accelerating the HMMER Sequence Analysis Suite Using Conventional Processors*

John Paul Walters
Institute for Scientific Computing
Wayne State University
Detroit, MI 48202
jwalters@wayne.edu

Bashar Qudah
Electrical and Computer Engineering
Wayne State University
Detroit, MI 48202
bqudah@wayne.edu

Vipin Chaudhary
Institute for Scientific Computing
Wayne State University
Detroit, MI 48202
vipin@wayne.edu

Abstract

Due to the ever-increasing size of sequence databases it has become clear that faster techniques must be employed to effectively perform biological sequence analysis in a reasonable amount of time. Exploiting the inherent parallelism between sequences is a common strategy. In this paper we enhance both the fine-grained and course-grained parallelism within the HMMER [2] sequence analysis suite. Our strategies are complementary to one another and, where necessary, can be used as drop-in replacements to the strategies already provided within HMMER. We use conventional processors (Intel Pentium IV Xeon) as well as the freely available MPICH parallel programming environment [1]. Our results show that the MPICH implementation greatly outperforms the PVM HMMER implementation, and our SSE2 implementation also lends greater computational power at no cost to the user.

1. Introduction

As the size of biological sequence databases continue to outpace processor performance, techniques must be found that can effectively process exponentially large databases. One common technique is to utilize reconfigurable hardware in order to accelerate the sequence analysis. DeCypher [15] is such a product. Another strategy that has been used with success is to use multicore network processors for bioinformatics processing [18]. While these techniques can be highly effective, they suffer from the fact that they make use of non-standard hardware. In this paper, we demonstrate that significant performance speedup can be achieved by utilizing widely available, off-the-shelf processors that are programmable via standard programming toolchains.

We implement our techniques within the HMMER suite, particularly the *hmmsearch* and *hmmpfam* programs. However, other sequence analysis options do exist. The Wise2 package specializes in comparing DNA sequences at the protein translation level [3]. SAM [16] is a set of tools utilizing linear hidden markov models, rather than the profile hidden markov models used by HMMER, to perform sequence analysis. PSI-BLAST [10] builds a position specific scoring matrix from each position in the alignment. The scoring matrix can then be used in further BLAST iterations and refined accordingly, resulting in greater sensitivity.

The remainder of this paper is organized as follows. In section 2 we present a review of the related work. In section 3 we give a brief overview of the HMMER sequence analysis suite. In section 4 we discuss the hardware used in our experiments. In section 5 we detail our SIMD/SSE2 implementation and in section 6 we discuss our MPICH/-cluster implementation. Our conclusions are detailed in section 7.

*This research was supported in part by NSF IGERT grant 9987598 and the Institute for Scientific Computing at Wayne State University.

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

2. Related Work

There have been several efforts to improve the speed and scalability of HMMER using both conventional hardware as well as porting the HMMER suite to more unique hardware platforms. We briefly discussed DeCypher [15] and the use of multicore network CPUs [18] in section 1. But other techniques also exist. ClawHMMER [5] is a GPU-based streaming Viterbi algorithm that purports to scale nearly linearly within a rendering cluster of Radeon 9800 Pro GPUs.

Erik Lindahl [9] has added an Altivec enhanced version of the P7Viterbi algorithm to the current HMMER code base. Lindahl makes use of the Altivec's included SIMD instructions to achieve speedup beyond that of the Intel Pentium IV Xeon.

3. Background

Most homology searches perform alignments in either a local or global fashion. The Smith/Waterman algorithm [13], for instance, is intended for local alignments while the Needleman/Wunsch algorithm [12] performs global alignments. The purpose of a global alignment is to find the similarities between two entire strings without regard to the specific similarities between substrings. A local alignment search, however, assumes that the similarities between two substrings may be greater than the similarities between the entire strings. In practice, local alignments are typically preferred.

Unlike typical homology searches, HMMER does not perform local or global alignments. Instead, the HMM model itself defines whether local or global alignment searches are performed. Typically, alignments are performed globally with respect to an HMM and locally with respect to a sequence [2].

HMMER is actually is not a single program, but rather a collection of several programs that perform different tasks to facilitate protein sequence analysis. Among the functionalities they provide are aligning sequences to an existing model, building a model from multiple sequence alignments, indexing an HMM database, searching an HMM database for matches to a query sequence, or searching a sequence database for matches to an HMM. The last two functionalities (i.e., the searches) are among the most frequently used and often require long execution times, depending on the input sequence or HMM and the size of database being searched against. These functionalities are provided by *hmmpfam* and *hmmsearch*, respectively.

Most of our experiments are performed on *hmmsearch* because it is more compute-bound than *hmmpfam*. That said, the techniques we present here also enhance the performance of *hmmpfam* as we demonstrate and discuss in section 5.

4. Hardware/Software Configuration

The experiments in this paper were performed on a university cluster. Each node is an SMP configuration consisting of two 2.66 GHz Pentium 4 Xeon processors with 2.5 GB of total system memory per node. 100 Mbit ethernet facilitates communication between each node. In addition, each node contains one PCI-X based Myrinet M3F-PCIXD-2 card with a data transfer rate of 2.12 Gbps.

Each node runs the Rocks v3.3.0 Linux cluster distribution. In addition, each node is loaded with both MPICH version 1.2.6 [1][4] and PVM version 3.4.3 [14]. All nodes are identical in every respect.

For testing purposes, most experiments were performed using the *nr* sequence database compared against *rrm.hmm* (*rrm.hmm* is included in the HMMER distribution). The *nr* database is 900 MB in size. A smaller version of the *nr* database was used to verify our results against smaller databases. To demonstrate the applicability of our *SSE2* optimizations in *hmmpfam* we also ran tests using the *Pfam* database.

5. SSE2

The *SSE2* [7] instructions are among a series of Intel *Single Instruction Multiple Data (SIMD)* extensions to the x86 Instruction Set Architecture (ISA). The first was the *MultiMedia eXtension (MMX)* which appeared in the Pentium MMX in 1997 [6]. MMX provides a series of packed integer instructions that work on 64-bit data using eight MMX 64-bit registers. MMX was followed by the *Streaming SIMD Extensions (SSE)* which appeared with Pentium III. SSE adds a series of packed and scalar single precision floating point operations, and some conversions between single precision floating point and integers. SSE uses 128-bit registers in a new XMM register file, which is distinct from the MMX register file. The *Second Streaming SIMD Extensions (SSE2)* appeared with the Pentium IV. SSE2 adds a series of packed and scalar double precision floating point operations, operating on 128-bit register files. SSE2 also adds a large number of data type conversion instructions. Finally, a third set of extensions, *SSE3* [8], was added to enable complex floating point arithmetic in several data layouts. *SSE3* also adds a small set of additional permutes and some horizontal floating point adds and subtracts.

As mentioned earlier, *HMMER* [2] is an implementation of *Profile hidden Markov models (profile HMMs)* for protein sequence analysis. Profile HMMs are statistical models of multiple sequence alignments. They capture position-specific information about how conserved each column of the alignment is, and which residues are likely. In general, profile HMMs can be used to perform sensitive database

searching using statistical descriptions of a sequence family's consensus.

Like other applications dealing with processing large arrays of data, HMMER has a strong potential to benefit from SIMD instructions by performing some of the time consuming operations in parallel on multiple data sets. However, re-implementing a relatively large application such as HMMER to take advantage of the newly added SIMD instructions is a costly and time consuming task. Further, moving from C (or any high level language) to assembly makes the code architecture-dependent rather than portable, and requires re-implementing the code for all supported platforms.

The more reasonable alternative is to limit the re-implementation to the smallest possible portion of code that results in the greatest speedup. We profiled HMMER and found that 15 lines of code consumed more than 50% of the execution time when *hmmpfam* or *hmmsearch* are used to perform a search. The 15-line section is simply a loop that performs some additions and maximum value selection over large arrays of 32-bit integers. SSE2, as described earlier, computes on 128-bit data and enables the programmer to perform several operations (e.g., addition) on four 32-bit integers in parallel.

However, the task is not as simple as vectorizing the previously mentioned 15 lines. Since the idea of SIMD is to perform an operation on 4 iterations (items) in parallel at the same time, the first problem is *inter-iteration dependencies*. That is, an operation in iteration i requires a result from iteration $i-1$ (or earlier iterations) to be performed. To resolve inter-iteration dependencies in our 15-line loop we had to split the loop into three loops. Each loop now iterates only 25% of the number of iterations in the original loop. We still achieve reasonable speedup, but not quite the ideal case as described above.

Splitting the loop is not the only overhead that can affect the overall reduction in execution time. We also encountered another problem: The lack of packed max/min instructions that works on 32-bit integers, similar to PMAXUB/PMINUB and PMAXSW/PMINSW that work on 8-bit and 16-bit data, respectively. Implementing a replacement for that missing instruction costs five SSE2 instructions for each occurrence. Assuming the data to be compared are initially in registers XMM3 and XMM4, where each register contains four integer items, and the maximum item of each pair is required to be in register XMM3 by the end of the task. If we have that "desired instruction" (let us call it PMAXD), the task can be performed simply by one instruction "PMAXD XMM4, XMM3" the replacement code is simply:

- MOVDQA XMM3, XMM5
  copying the content of XMM3 into XMM5
- PCMPGTD XMM4, XMM5
  Comparing contents of XMM4 and XMM5 and for each pair, if the item in XMM4 is greater than that in XMM5, the item in XMM5 is replaced with 0's, otherwise it is replaced by all 1's. By the end of this step, each of the four items in XMM5 will be either 0x00000000 or 0xFFFFFFFF. The original data in XMM5 are lost and that is why we copied them in the previous step.
- PAND XMM5, XMM3
  Bitwise AND the content of the two registers and put the results in XMM3. Since XMM3 has the same contents as those of XMM5 before the previous step, this step will keep only the maximum values in XMM3 and replace those which are not the maximum in their pairs by 0's.
- PANDN XMM4, XMM5
  Invert XMM5 (1's complement) and AND it with XMM4. That will have a similar result as in the previous step but the maximum numbers in XMM4 will be stored in XMM5 this time.
- POR XMM5, XMM3
  This will gather all the maximums in XMM5 and XMM3 and store them in XMM3. The task is done.

Fortunately, even with these five instructions replacing the desired instruction, we can still achieve reasonable speedup over the non-SSE2 case. With no SIMD, the maximum selection consists of three instruction: compare, jump on a condition, then a move instruction which will be executed only if the condition fails. Assuming equal probabilities for the fail and the success of the condition, that means an average of 2.5 instructions for each pair of items. That is 10 instructions for four pairs compared to the five when the SSE2 instructions are used.

We should note that the Altivec architecture provides the needed instruction in the form of VMAXSW and VMAXUW (vector max signed/unsigned max). This is used in the Erik Lindahl [9] port to achieve excellent speedup on the PowerPC architecture.

Finally, an additional overhead is shared, typically by several SSE2 instructions: that is, data alignment and the moving of data into the 128-bit XMM registers. However, once the data is in these registers, many SSE2 operations can be performed on them, assuming efficiently written code and that the entire alignment and loading cost can be shared. Even if this is not the case some speedup can still be observed over the non SIMD case,

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

5.1. SSE2 Evaluation

We begin our evaluation by noting that approximately 50% of the runtime of our code can be vectorized using the *SSE2* instructions. We can therefore use Amdahl's law to compute the theoretical maximum speedup possible given 50% parallelizable code. We start from Amdahl's law:

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \quad (1)$$

From equation 1 we have P = the percentage of the code that can be parallelized. $1-P$ is therefore the percentage of code that must be executed serially. Finally, N from equation 1 represents the number of processors. In this case, N actually represents the number of elements that can be executed within a single SSE2 instruction, 4.

Theoretically, the expected speedup of those 15 lines is 4, This should therefore result in an expected speedup of

$$Speedup = \frac{1.0}{50\% + \frac{50\%}{4}} = 1.6 \quad (2)$$

In other words the overall reduction in execution time is expected to be:

$$1 - \frac{1}{1.6} = 37.5\% \quad (3)$$

Our analysis shows a reduction in the execution time even considering the overhead described in section 5. The 15 lines of code were re-implemented using the SSE2 instructions and *hmmpfam* and *hmmsearch* were used to compare the results. Many samples were used in searches against the *Pfam* and *nr* databases[17][11]. The Pfam database is a large collection of multiple sequence alignments and hidden Markov models covering many common protein families. The *nr* database, is a non-redundant database available from [11]. Each search was repeated several times and the average was found for each search both when the original code is used, and when the modified code using SSE2 is used. The reduction in execution time varies from around 18% up to 23% depending on the sample and the percentage of time spent in the re-implemented code. Table 1 shows the results for three samples. Samples 1 and 2 were taken from *hmmpfam* searches while sample 3 was taken from *hmmsearch* searches. The corresponding speedups are from around 1.2 up to 1.3.

Implementing more code using SSE2 may have resulted in greater speedup, but would have been a much more costly task. The advantage to this speedup is that it's cost free, no new hardware is required, and no real development time is needed. Only a small portion of the code needs to be re-implemented and maintained over the original implementation.

Table 1. Effect of SSE2 on HMMER Execution Time

	Average Execution Time (seconds)		Reduction in
	Original Code	**with SSE2**	Execution Time
Sample 1	1183	909	23.2%
Sample 2	272	221	18.8%
Sample 3	1919	1562	18.6%

6. Cluster/MPI Parallelism

In this section we describe our HMMER MPI implementation. Unlike the SIMD/SSE2 implementation, the MPI implementation takes advantage of the parallelism between multiple sequences, rather than the instruction level parallelism used by the SSE2 technique. The advantage in this case is that greater parallelism can be achieved by offloading the entire *P7Viterbi()* function to compute nodes, rather than simply vectorizing the 15 most time consuming lines.

6.1. Parallelizing the Database

Rather than the instruction-level parallelism described in section 5, we now distribute individual sequences to cluster nodes. Each cluster node then performs the majority of the computation associated with its own sequence and returns the results to the master node. This is the method by which the original PVM implementation of HMMER performs the distribution. It is also the basis from which we began our MPI implementation. The important point to note is that a single function, *P7Viterbi()*, accounts for greater than 90% (see table 2) of the runtime. Therefore it is imperative that it be executed on the worker nodes if any effective parallelism is to be achieved.

6.2. Enhancing the Cluster Distribution

While the strategy demonstrated above does indeed yield reasonable speedup, we found that the workers were spending too much time blocking for additional work. The solution to this problem is twofold. The workers should be using a non-blocking, double buffering strategy rather than their simple blocking techniques. Second, the workers can reduce the communication time by processing database chunks rather than individual sequences.

Our double buffering strategy is to receive the next sequence from the master node while the current sequence is being processed. The idea behind double buffering is to overlap as much of the communication as possible with the computation, hopefully hiding the communication altogether.

In keeping with the strategy used in the PVM implementation, the master does not also act as a client itself. Instead, its job is to supply sequences as quickly as possible to the workers as newly processed sequences arrive. Therefore, a cluster of N nodes will actually have only $N - 1$ worker nodes available with one node reserved as the master.

While double buffering alone improved the speedup immensely, we also sought to reduce the communication time in addition to masking it through double buffering. To this end we simply bundled several sequences (12, in our experiments) to each worker in each message. We settled on 12 sequences by simply observing the performance of *hmmsearch* for various chunk sizes. Sending 12 sequences in each message maintained a reasonable message size and also provided enough work to keep the workers busy while the next batch of sequences was in transit.

6.3. MPI Results

Beginning from equation 1, we can derive a formula for the expected speedup of *hmmsearch* for a given number of CPUs. Table 2 lists our results of the profile.

Table 2. Profile results of *hmmsearch*

Function	% of total execution
P7Viterbi	97.72
P7ViterbiTrace	0.95
P7ReverseTrace	0.25
addseq	0.23
other	0.85

We notice that the *P7Viterbi* function accounts for nearly all of the runtime of *hmmsearch*. Furthermore, of the functions listed in table 2 the first 3 are all run on the worker node. Therefore, our P from equation 1 can be reasonably approximated as 98.92%. For two worker processors, this leaves us with an expected speedup of 1.98 with an expected decrease in execution time of 49%.

Table 3. Actual Speedup compared to Optimal Speedup (non-*SSE2*)

N CPU	Actual Speedup	Optimal Speedup
1	1	1
2	1.62	1.98
4	3.09	3.87
8	6.44	7.44
16	11.10	13.77

From table 3 we can see that the actual speedup of 2 CPUs is 1.62 or approximately a 38% decrease in run time. Considering that the implementation requires message passing over a network and that the messages and computation cannot necessarily be overlapped entirely, we feel that the actual speedup is reasonable.

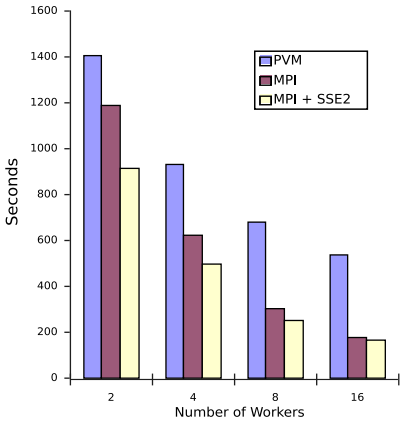

Figure 1. Comparative timings of *PVM, MPI, MPI+SSE2* implementations

In figure 1 we provide our raw timings for *hmmsearch*, comparing our *MPI* and *MPI+SSE2* code against the *PVM* code provided by the *HMMER* source distribution. In figure 2 we translate the numbers from figure 1 into their corresponding speedups and compare them against one another.

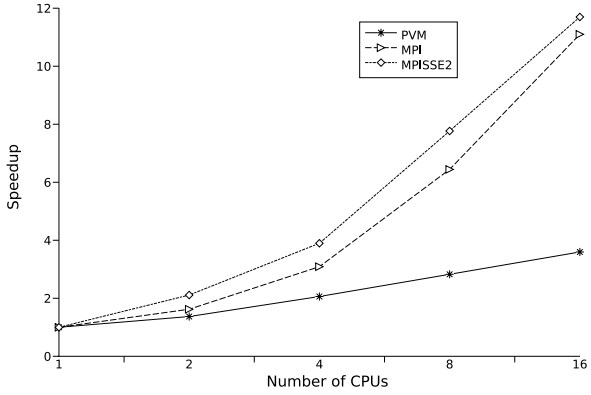

Figure 2. Figure 1 translated into the corresponding speedups

To verify that our techniques work in the case of smaller databases, we also tested *hmmsearch* with a smaller (100 MB) version of the *nr* database. The smaller database was created by simply taking the first 100 MB of *nr*. Our results are summarized in table 4. From table 4 we can see that both the *MPI* and the *SSE2* techniques yield reasonable speedup from even fairly small databases. By examining figure 2

and table 4 we can also see that our speedup increases with larger databases.

Table 4. Speedups of *hmmsearch* for 100 MB database

# CPU	PVM	MPI	MPI+SSE2
2	1.39	1.69	2.21
4	2.28	3.38	3.84
8	4.05	5.81	6.65
16	4.56	5.90	7.71

As can be seen in Figure 1, our MPI implementation clearly outperforms the PVM implementation by a fairly wide margin. As the number of of nodes increases, the MPI implementation improves the runtime by nearly a factor of two. And adding SSE2 improves upon the MPI implementation. Figure 2 clearly shows that our MPI implementation scales much better than the current PVM implementation. In addition, some of the speedup may be due, at least in part, to the underlying differences between PVM and MPI.

We should also note that tests were run against the *nr* database using the Myrinet interfaces on our cluster nodes. In this case we saw no run time advantage to using the Myrinet interfaces.

7. Conclusions

We have improved upon the *HMMER* sequence analysis suite by implementing the core of the algorithm using the Intel *SSE2* instruction set. We have also demonstrated large improvement in the clustered implementation of *HMMER* by porting the client and server to use MPI rather than PVM. Furthermore, our MPI implementation utilized an effective double buffering and database chunking strategy to provide performance increases beyond that which would be achieved by directly porting the PVM code to MPI code. Our results show excellent speedup over and above that of the PVM implementation. Our implementation makes use of standard hardware and thus can be used as a drop-in replacement for anyone wishing to accelerate *HMMER* searches on their own Pentium IV cluster.

Acknowledgments

We would like to acknowledge the generous help of Joe Landman from Scalable Informatics, LLC[1]. Joe's expertise was particularly helpful in the preliminary portions of this research.

[1] http://www.scalableinformatics.com

References

[1] MPICH-A Portable Implementation of MPI. http://www-unix.mcs.anl.gov/mpi/mpich/.

[2] S. Eddy. HMMER: profile HMMs for protein sequence analysis. http://hmmer.wustl.edu.

[3] EMBL-EBI. Wise2: Intelligent algorithms for DNA searches. http://www.ebi.ac.uk/Wise2/.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[5] D. R. Horn, M. Houston, and P. Hanrahan. Clawhmmer: A streaming hmmer-search implementation. In *To appear in SC '05: The International Conference on High Performance Computing, Networking and Storage*, 2005.

[6] Intel Corporation. MMX: MultiMedia eXtensions. http://www.intel.com.

[7] Intel Corporation. SSE2: Streaming SIMD (Single Instruction Multiple Data) Second Extensions. http://www.intel.com.

[8] Intel Corporation. SSE3: Streaming SIMD (Single Instruction Multiple Data) Thiord Extensions. www.intel.com.

[9] E. Lindahl. Altivec-accelerated HMM algorithms. http://lindahl.sbc.su.se/.

[10] NCBI. Position-specific iterated BLAST. http://www.ncbi.nlm.nih.gov/BLAST/.

[11] NCBI. The NR (non-redundant) database. ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz.

[12] S. Needleman and C. Wunsch. A general method applicable to the search for similarities in the amino acid sequence of two sequences. *J. Mol. Biol.*, 48(3), 1970.

[13] T. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *J. Mol. Biol.*, 147, 1981.

[14] V. S. Sunderam. PVM: a framework for parallel distributed computing. *Concurrency: Pract. Exper.*, 2(4):315–339, 1990.

[15] TimeLogic bioComputing solutions. DecypherHMM. http://www.timelogic.com/.

[16] UCSC Computational Biology group. Sequence alignment and modeling system. http://www.cse.ucsc.edu/research/compbio/sam.html.

[17] Washington University in St. Louis. The PFAM HMM library: a large collection of multiple sequence alignments and hidden markov models covering many common protein families. http://pfam.wustl.edu.

[18] B. Wun, J. Buhler, and P. Crowley. Exploiting coarse-grained parallelism to accelerate protein motif finding with a network processor. In *PACT '05: Proceedings of the 2005 International Conference on Parallel Architectures and Compilation Techniques*, 2005.