1 Optimized Cluster-Enabled HMMER Searches

John Paul Walters,[†] Joseph Landman[‡] and Vipin Chaudhary[§]

1.1 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 [Eddy, 1998] and coworkers [Durbin et al., 1998]. 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 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 [Lindahl, 2005], a GPU port of *hmmsearch* by Horn et al. of Stanford [Horn et al., 2005] as well as several optimizations performed by the authors of this chapter. These optimizations 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. Detailed descriptions of the author's modifications can also be found in [Walters et al., 2006, Landman et al., 2006].

The remainder of this chapter is organized as follows: in section 1.2 we give a brief overview of HMMER and the underlying plan-7 architecture. In section 1.3 we discuss several different strategies that have been used to implement and accelerate HMMER on a variety of platforms. In section 1.4 we detail our optimizations and provide performance details. We conclude this chapter in section 1.5

[†]Institute for Scientific Computing, Wayne State University, Detroit, MI
[‡]Scalable Informatics LLC, Canton, MI
[§]Department of Computer Science and Engineering, University at Buffalo, The State University of New York, Buffalo NY

1.2 BACKROUND

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 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 [Eddy, 1998] architecture. The profile HMM architecture is constructed using the plan-7 model as depicted in figure 1.1.

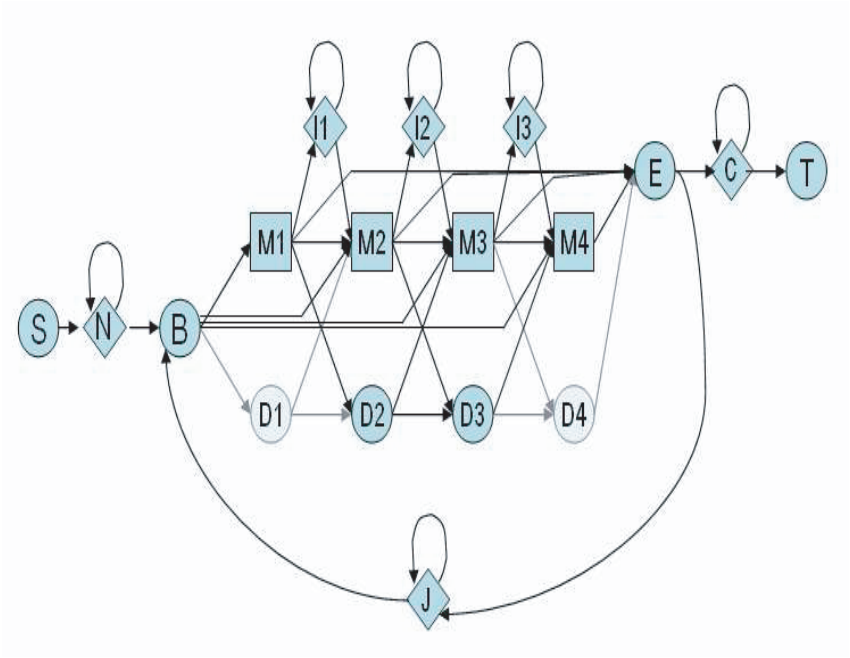

Fig. 1.1 Plan 7 HMM model architecture

This architecture encodes insert, deletion, and match states all relative to a consensus sequence model. The plan-7 architecture is a Viterbi algorithm [Viterbi, 1967] and the code implementing the plan-7 architecture is constructed as such. Viterbi algorithms involve state vector 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 model. 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.

Most homology searches perform alignments in either a local or global fashion. The Smith/Waterman algorithm [Smith and Waterman, 1981], for instance, is intended for local alignments while the Needleman/Wunsch algorithm [Needleman and Wunsch, 1970] 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 [Eddy, 2006].

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.

1.3 TECHNIQUES FOR ACCELERATING HMMER

As we mentioned in section 1.1, there have been a variety of techniques used to both implement and accelerate HMMER searches. They range from typical high performance computing (HPC) strategies such as clustering, to web services, and even extending the core HMMER algorithm to novel processing architectures. In this section we discuss in greater depth the various strategies used in both implementing and accelerating HMMER.

1.3.1 Network and Graphics Processors

1.3.1.1 **JackHMMer** We begin with a discussion of *JackHMMer* [Wun et al., 2005] where network processors are used in place of a general purpose processor in order to accelerate the core Viterbi algorithm. Specifically, *JackHMMer* uses the Intel IXP 2850 network processor. Like many network processors, the Intel IXP 2850 is a heterogeneous multicore chip. This is, several different processing elements are integrated into a single chip. In this case, 16 32-bit microengines (MEs) are paired with a single XScale ARM-compatible processor. Each microengine run at 1.4 GHz while the XScale CPU runs at a peak rate of 700 MHz. Other processing elements, such as memory controllers and interconnect also run at 700 MHz.

JackHMMer essentially uses the IXP 2850 as a single-chip cluster with the XScale CPU functioning as the head node. Like a typical cluster, the XScale CPU is

responsible for distributing jobs to the individual microengines. In *JackHMMer*, each job takes the form of a *Viterbi packet*. A database of HMMs is divided into a series of *Viterbi packets* with each packet corresponding to an individual database model [Wun et al., 2005]. The XScale processor distributes the *Viterbi packets* to the microengines where each microengine then independently performs the Viterbi algorithm on the packet. In the standard HMMER implementation, this computation is performed by hmmpfam.

Despite the IXP's apparently slow clock speed, the authors of [Wun et al., 2005] claim a speedup of 1.82x compared to a P4 running at 2.6 GHz. However, it is important to note that the algorithm implemented in *JackHMMer* is not the full Viterbi algorithm as implemented in HMMER. A small amount of processing time is saved by not computing the post-processing portion of the HMMER reference implementation. In addition, Wun et al. note that up to 25% of the initial time was spent in a pre-processing stage in which HMM models are converted into log-odds form as required by the Viterbi algorithm. Instead, they precompute the log-odds data ahead of time and store it on disk for future use. This technique could also be used in a standard HMMER implementation.

1.3.1.2 **ClawHMMER** A second technique that has gained prominence in sequence analysis is the use of streaming processors/graphics processors. While actual streaming processors are not yet widely available, graphics processors bear a close resemblance with regard to functionality. Unlike traditional general purpose processors, graphics hardware has been optimized to perform the same operation over large streams of input data. This is similar to the SIMD operations of general purpose CPUs, but with greater width and speed.

Unlike the SIMD approach to optimizing HMMER, *ClawHMMER* [Horn et al., 2005] operates over multiple sequences rather than vectorizing the computation of individual sequences. The key to *ClawHMMER*'s speed is that sequences many sequences are computed on simultaneously. The time to process a group of sequences is essentially the time to process the longest sequence in the batch. Therefore, is it advantageous to group sequences into similarly-sized chunks (based on the sequence length). Unlike *JackHMMer*, *ClawHMMER* implements the *hmmsearch* function of the standard HMMER implementation.

In [Horn et al., 2005] Horn et al. demonstrate the speed of *ClawHMMER* with an implementation of their streaming Viterbi algorithm on a 16 node rendering cluster. The cluster consisted of 16 nodes with a Radeon 9800 Pro GPU in each node. Since each sequence is independent of the others, the Viterbi algorithm is highly parallel. Thus, Horn et al. are able to demonstrate nearly linear speedup with their streaming Viterbi algorithm.

1.3.2 *DecypherHMM*

For the fastest possible sequence analysis, a custom processor is a necessity. Typically, such customized hardware comes in the form of an FPGA (field programmable gate array). Historically, FPGAs have been difficult and time-consuming to program.

They require expertise in hardware/CPU design and are quite costly. However, they are often able to achieve 10-100x the speed of a general purpose CPU or cluster.

Timelogic provides an FPGA HMM protein characterization solution named *DeCypherHMM* [TimeLogic BioComputing solutions, 2006]. The DeCypher engine is deployed as a standard PCI card into an existing machine. Multiple DeCypher engines can be installed in a single machine, which according to TimeLogic, results in near linear speedup.

1.3.3 Web Services

Web based sequence analysis tools are becoming popular for all areas of bioinformatics research. In this section we detail two of the most popular web based toolkits for facilitating HMMER searches, *SledgeHMMER* and the *MPI Bioinformatics Toolkit*.

1.3.3.1 **SledgeHMMER** *SledgeHMMER* [Chukkapalli et al., 2004] is a web service designed to allow researches to perform Pfam database searches without having to install HMMER locally. To use the service, a user submits a batch job to the *SledgeHMMER* website. Upon completion of the job, the results are simply emailed back to the user

In addition to being available via the web, *SledgeHMMER* also includes three optimizations to expedite Pfam searches. The first optimization is their use of precalculated search results. Those queries that match entries held within the *SledgeHMMER* database can be quickly returned. Matching entries are found using an MD5 hashing strategy.

For those results that are not already contained within the *SledgeHMMER* database, *SledgeHMMER* uses a parallelized *hmmpfam* algorithm. Rather than using MPI or PVM to perform the distributed search, *SledgeHMMER* relies on a Unix-based file-locking strategy. This allows nodes to leave/join as they become available, but also requires a shared filesystem from which all nodes access a lock-file. This lock-file acts as an iterator and returns indexes which correspond to query sequences. By using a lock-file, *SledgeHMMER* ensures that all sequences are distributed exactly once.

The final optimization employed by *SledgeHMMER* is to read the entire Pfam database into memory before performing the batch search. In a typical scenario the entire Pfam database will be read for each query sequence in the batch search. This can be extremely time-consuming. To alleviate this problem, the Pfam database is read and stored into memory upon start up and can be referenced throughout the computation without accessing the disk.

1.3.3.2 **The MPI Bioinformatics Toolkit** The *MPI*(Max-Planck-Institute) *Bioinformatics Toolkit* [Biegert et al., 2006] is a web-based collection of bioinformatics tools that is freely accessible to researchers. The toolkit makes two major contributions to web-based bioinformatics services.

The first contribution is the toolkit's vast collection of tools, all of which are available from a single website. These tools not only include HMMER searches, but

BLAST [Altschul et al., 1990, Altschul et al., 1997], ClustalW [Thompson et al., 1994], and MUSCLE[Edgar, 2004] searches (among others) in addition to many tools developed in-house. Some of these tools, specifically HMMER, include optimizations to accelerate the searching of sequences. In the case of HMMER, the *MPI Bioinformatics Toolkit* reduces HMMER searches to $\sim$10% of their original. This is done by reducing the database with a single iteration of PSI-BLAST.

The second major contribution made by the *MPI Bioinformatics Toolkit* is to allow the user to pipeline searches from one tool to another automatically. This allows the results of an initial search to be fed into a secondary search (for example, from a sequence analysis tool to a formatting tool or classification tool). Furthermore, a user can store customized databases on the *MPI Bioinformatics Toolkit* server for future use.

1.4 CONVENTIONAL CPU OPTIMIZATIONS

In this section we detail three optimizations made by the authors of this chapter. In the first case, we evaluate changes made through absolute minimal changes in source code. In this case, the changes were designed to allow the compiler to perform its optimizations in a more efficient manner. Such changes also benefit from portability as well. Our second strategy was to manually add SSE2 code to the P7Viterbi function. This required the addition of inline assembly code resulting in nonportable code, but accelerated code. Our final strategy was to recast the computation in terms of MPI such that multiple nodes could be used simultaneously. The MPI implementation is portable across standards-compliant MPI implementations.

1.4.1 Hardware/Software Configuration

The experiments in this chapter 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.

Each node runs the Rocks v3.3.0 Linux cluster distribution. In addition, each node is loaded with both MPICH version 1.2.6 [Argonne National Lab, 2006] [Gropp et al., 1996] and PVM version 3.4.3 [Sunderam, 1990]. 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.

In section 1.4.2, tests were performed using BBSv3 [Landman, 2006] tests on a 64-bit AMD Opteron. The binaries were compiled with threading disabled.

1.4.2 Minimal Source Changes

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 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 listing 1.1 to that of listing 1.2.

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 $sc1$ 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.

1.4.2.1 Performance Results of the Minimal Changes

The binary generated by 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. The test binary used *hmmcalibrate* to test the efficacy of the Viterbi improvements.

From figure 1.2 we can see that modest improvements can be gained from seemingly minor source code changes. Such changes, while small, can have a dramatic affect on the compiler's ability to optimize the source code. In cases where most of

Listing 1.1 The most time consuming portion of the P7Viterbi algorithm

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

  dc[k] = dc[k-1] + tpdd[k-1];
  if ((sc = mc[k-1] + tpmd[k-1]) > dc[k])
    dc[k] = sc;
  if (dc[k] < -INFTY)
    dc[k] = -INFTY;

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

the compute-time is spent in a single function, such compiler optimizations can be particularly useful.

1.4.3 Inline Assembly/SSE2

The *SSE2* [Intel Corporation, 2006b] 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 [Intel Corporation, 2006a]. 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

Listing 1.2 Removing the conditional from the innermost loop

```
for (k = 1; k < M; k++) {
  ...
  ...
  ...

    ic[k] = mpp[k] + tpmi[k];
    if (( ip[k] + tpii[k]) > ic[k])
      ic[k] = ip[k] + tpii[k];
    ic[k] += is[k];
    if (ic[k] < -INFTY)
      ic[k] = -INFTY;

}

    k = M;
    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;

    dc[k] = dc[k-1] + tpdd[k-1];
    if (( mc[k-1] + tpmd[k-1]) > dc[k])
        dc[k] = mc[k-1] + tpmd[k-1] ;
    if (dc[k] < -INFTY) dc[k] = -INFTY;
```

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. In addition, SSE2 provides integer operations similar to those available with MMX except that they work on 128-bit data and use the XMM register file. SSE2 also adds a large number of data type conversion instructions. More recently, a third set of extensions, *SSE3* [Intel Corporation, 2003], 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.

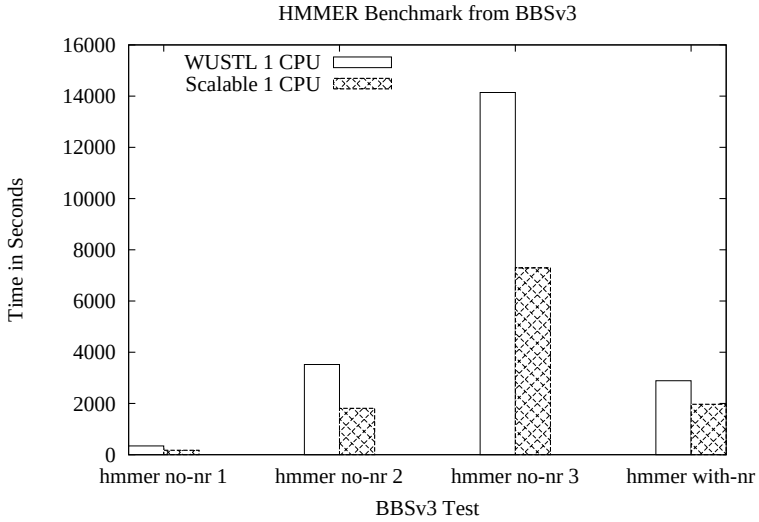

Fig. 1.2 BBSv3 Results

Like other applications dealing with processing large arrays of data, HMMER seemingly has a strong potential to benefit from SIMD instructions by performing some of the time consuming operations in parallel on multiple data sets. Recall that a similar strategy was discussed in section 1.3.1.2 with regard to graphics processors. 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. In our profile of HMMER we found that the innermost loop of the Viterbi function (see listing 1.1) consumed more than 50% of the execution time when *hmmpfam* or *hmmsearch* are used to perform a search. This short code segment is simply a loop that performs some additions and maximum value selections 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. Ideally this would lead to a $4x$ speedup in the vectorized code segment.

However, the task is not as simple as vectorizing the previously mentioned code segment. 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 loop we had to split the loop into three loops. That may appear to add additional overhead, but 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 instructions: 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 [Lindahl, 2005] 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,

1.4.3.1 SSE2 Evaluation and Performance Results 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:

$$Speedup = \frac{1}{(1-P) + \frac{P}{N}} \tag{1.1}$$

From equation 1.1 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 1.1 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 the loop is 4, This should therefore result in an expected speedup of

$$Speedup = \frac{1.0}{50\% + \frac{50\%}{4}} = 1.6 \tag{1.2}$$

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

$$1 - \frac{1}{1.6} = 37.5\% \tag{1.3}$$

Our analysis shows a reduction in the execution time even considering the overhead described in section 1.4.3. The loop was then 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 [Pfam, 2006][NCBI, 2006]. 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 [NCBI, 2006]. 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 24% depending on the sample and the percentage of time spent in the re-implemented code. Table 1.1 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 the SSE2 may have resulted in more speedup, but would have been a much more costly task. The advantage to this speedup is that it is cost-free, no new hardware is required, and no real development time is needed, just a small portion of the code needs to be re-implemented and maintained over the original implementation. This disadvantage is the lack of portability.

Table 1.1 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%

1.4.4 Cluster/MPI Parallelism

In this section we describe our HMMER MPI implementation. Unlike the SIMD/SSE2 and minimal source-change implementations, 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 most time consuming loop.

1.4.4.1 Parallelizing the Database Rather than the instruction-level parallelism described in section 1.4.3, 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. To understand the distribution of computation between the master node and the worker nodes, we provide pseudocode in listings 1.3, 1.4, and 1.5. The important point to note is that the *P7Viterbi()* function accounts for greater than 90% (see table 1.2) of the runtime, thus is is imperative that it be executed on the worker nodes if any effective parallelism is to be achieved.

1.4.4.2 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

Listing 1.3 Pseudocode of each sequence iteration

```
while (ReadSeq(...)){
  dsq = DigitizeSequence(...);
  if (do_xnu && Alphabet_type
          == hmmAMINO)
    XNU(...);
  sc = P7Viterbi(...);
  if (do_forward) {
    sc  = P7Forward(...);
    if (do_null2)
      sc -= TraceScoreCorrection(...);
  }
  pvalue = PValue(hmm, sc);
  evalue = thresh->Z ?
        (double) thresh->Z * pvalue :
        (double) nseq * pvalue;
  if (sc >= thresh->globT &&
        evalue <= thresh->globE){
    sc = PostprocessSignificantHit(...);
  }
  AddToHistogram(histogram, sc);
}
```

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 rreasonablemessage size and also provided enough work to keep the workers busy while the next batch of sequences was in transit.

1.4.4.3 MPI Performance Results Beginning from equation 1.1, we can derive a formula for the expected speedup of *hmmsearch* for a given number of CPUs. For example, let us assume that the number of CPUs is 2. From equation 1.1 we can

Listing 1.4 Pseudocode of the master node

```
while (ReadSeq(...)){
      /* receive output */
  pvm_recv(slave_tid, HMMPVM_RESULTS);
      /* send new work */
  dsq = DigitizeSequence(...);
  if (do_xnu) XNU(...);
  pvm_send(slave_tid, HMMPVM_WORK);

      /*process output */
  if (sent_trace){
    sc = PostprocessSignificantHit(...);
  }
  AddToHistogram(...);
}
```

express the potential speedup as:

$$\frac{1}{(1-P)+\frac{P}{2}} \tag{1.4}$$

Again, P is the percentage of code executed in parallel and $(1-P)$ is the serial code. In order to find the fraction of code capable of being parallelized we profiled *hmmsearch* using the *nr* database. Table 1.2 lists our results of the profile.

Table 1.2 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 1.2 the first 3 are all run on the worker node. Therefore, our P from equation 1.4 can be reasonably approximated as 98.92%. For two worker processors, this leaves us with an expected speedup of

$$\frac{1}{(1-.9892)+\frac{.9892}{2}} = 1.98 \tag{1.5}$$

Listing 1.5 Pseudocode of the worker node

```
for (;;){
/*receive work*/
  pvm_recv(master_tid, HMMPVM_WORK);

/*compute alignment*/
  sc = P7Viterbi(...);
  if (do_forward) {
    sc  = P7Forward(...);
    if (do_null2)
      sc -= TraceScoreCorrection(...);
  }

  pvalue = PValue(...);
  evalue = Z ? (double) Z * pvalue :
               (double) nseq * pvalue;
  send_trace = (tr != NULL &&
               sc >= globT
               && evalue <= globE)
               ? 1 : 0;

      /* return output
       */

  if (send_trace) PVMPackTrace(...);
  pvm_send(master_tid, HMMPVM_RESULTS);
}
```

with an expected increase in exection time of 49%.

Table 1.3 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 1.3 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 rather respectable.

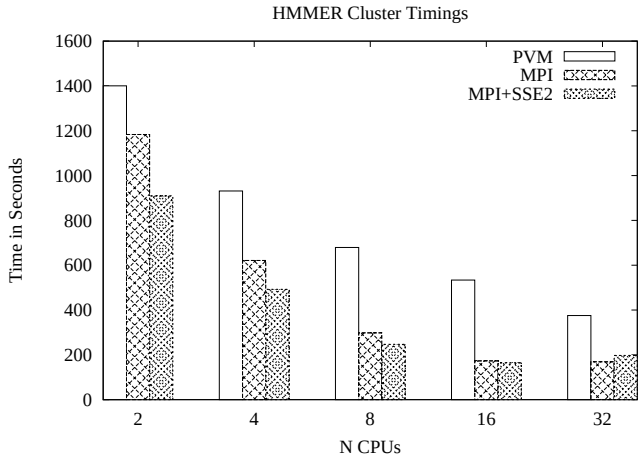

Fig. 1.3 Comparative timings of *PVM, MPI, MPI+SSE2* implementations

In figure 1.3 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 table 1.4 we translate the numbers from figure 1.3 into their corresponding speedups and compare them against one another.

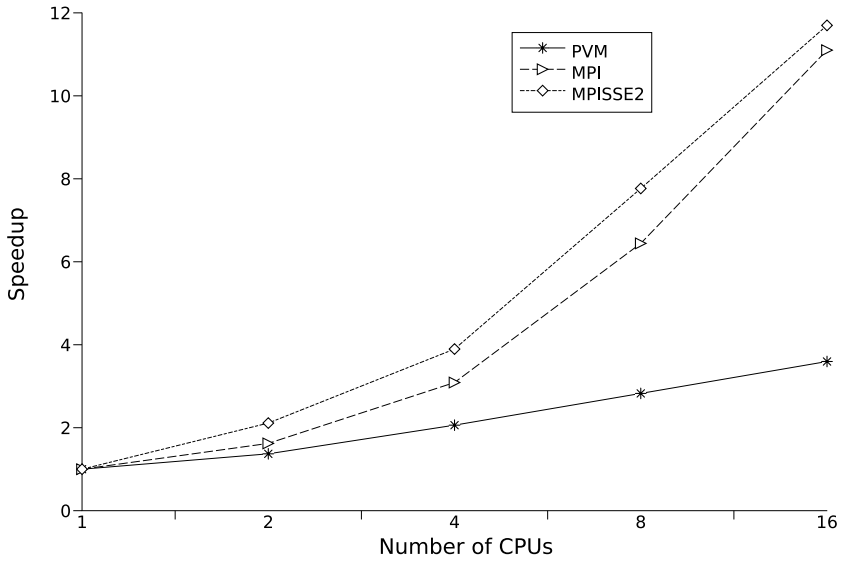

Fig. 1.4 Figure 1.3 translated into the corresponding speedups

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 1.4. From table 1.4 we can see that both the *MPI* and the *SSE2* techniques yield reasonable speedup from even fairly small databases. By examining figure 1.4 and table 1.4 we can also see that our speedup increases with larger databases.

Table 1.4 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 1.3, 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 1.4 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.

1.5 CONCLUSIONS

We have discussed the many ways in which HMMER has been accelerated and improved through a variety of mechanisms. These include novel hardware solutions, web services and conventional CPU acceleration techniques. Owing to its popularity, *HMMER* has inspired a wealth of freely available web services that enable the linking of multiple sequence analysis tools into a single service. We have also shown how *HMMER* can be effectively accelerated on typical hardware in order to more effectively utilize the resources that are already available. Our acceleration strategies ranged from minimal source code changes to inline assembly and MPI. We have further demonstrated large improvements 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 and beyond that of any conventional hardware. The techniques that we have implemented could prove useful beyond the cluster implementation. Indeed even the web services will soon find the need

to cluster enable-their computational backends. Our accelerations could therefore easily be used within a web-based sequencing package for high scalability.

REFERENCES

Altschul et al., 1990. Altschul, S. F., Gish, W., Miller, W., Myers, E. W., and Lipman, D. J. (1990). Basic local alignment search tool. *J Mol Biol*, 215(3):403–410.

Altschul et al., 1997. Altschul, S. F., Madden, T. L., SchÃffer, A. A., Zhang, J., Zhang, Z., Miller, W., and Lipman, D. J. (1997). Gapped blast and psi-blast: a new generation of protein database search programs. *Nucleic Acids Res*, 25(17):3389–3402.

Argonne National Lab, 2006. Argonne National Lab (2006). MPICH-A Portable Implementation of MPI. `http://www-unix.mcs.anl.gov/mpi/mpich/`.

Biegert et al., 2006. Biegert, A., Mayer, C., Remmert, M., Soding, J., and Lupas, A. (2006). The MPI bioinformatics toolkit for protein sequence analysis. *Nucleic Acids Research*, 34(Web Server issue).

Chukkapalli et al., 2004. Chukkapalli, G., Guda, C., and Subramaniam, S. (2004). SledgeHMMER: a web server for batch searching the pfam database. *Nucleic Acids Research*, 32(Web Server issue).

Durbin et al., 1998. Durbin, R., Eddy, S., Krogh, A., and Mitchison, A. (1998). *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press.

Eddy, 1998. Eddy, S. (1998). Profile hidden Markov models. *Bioinformatics*, 14(9).

Eddy, 2006. Eddy, S. (2006). HMMER: profile HMMs for protein sequence analysis. `http://hmmer.wustl.edu`.

Edgar, 2004. Edgar, R. C. (2004). Muscle: multiple sequence alignment with high accuracy and high throughput. *Nucleic Acids Res*, 32(5):1792–1797.

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

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

Intel Corporation, 2003. Intel Corporation (2003). SSE3: Streaming SIMD (Single Instruction Multiple Data) Thiord Extensions. `www.intel.com`.

Intel Corporation, 2006a. Intel Corporation (2006a). MMX: MultiMedia eXtensions. http://www.intel.com.

Intel Corporation, 2006b. Intel Corporation (2006b). SSE2: Streaming SIMD (Single Instruction Multiple Data) Second Extensions. http://www.intel.com.

Landman, 2006. Landman, J. (2006). Bbsv3. http://www.scalableinformatics.com/bbs.

Landman et al., 2006. Landman, J., Ray, J., and Walters, J. P. (2006). Accelerating hmmer searches on opteron processors with minimally invasive recoding. In *AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 2 (AINA'06)*, pages 628–636, Washington, DC, USA. IEEE Computer Society.

Lindahl, 2005. Lindahl, E. (2005). Altivec-accelerated HMM algorithms. http://lindahl.sbc.su.se/.

NCBI, 2006. NCBI (2006). The NR (non-redundant) database. ftp://ftp.ncbi.nih.gov/blast/db/FASTA/nr.gz.

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

Pfam, 2006. Pfam (2006). The PFAM HMM library: a large collection of multiple sequence alignments and hidden markov models covering many common protein families. http://pfam.wustl.edu.

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

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

Thompson et al., 1994. Thompson, J. D., Higgins, D. G., and Gibson, T. J. (1994). Clustal w: improving the sensitivity of progressive multiple sequence alignment through sequence weighting, position-specific gap penalties and weight matrix choice. *Nucleic Acids Res*, 22(22):4673–4680.

TimeLogic BioComputing solutions, 2006. TimeLogic BioComputing solutions (2006). DecypherHMM. http://www.timelogic.com/.

Viterbi, 1967. Viterbi, A. J. (1967). Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Trans. Inform. Theory*, IT-13:260–269.

Walters et al., 2006. Walters, J. P., Qudah, B., and Chaudhary, V. (2006). Accelerating the hmmer sequence analysis suite using conventional processors. In

AINA '06: Proceedings of the 20th International Conference on Advanced Information Networking and Applications - Volume 1 (AINA'06), pages 289–294, Washington, DC, USA. IEEE Computer Society.

Wun et al., 2005. Wun, B., Buhler, J., and Crowley, P. (2005). 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*.