2018 IEEE International Parallel and Distributed Processing Symposium Workshops

Modifying HMMER3 to Run Efficiently on the Cori Supercomputer using OpenMP Tasking

William Arndt
National Energy Research Scientific Computing Center
Lawrence Berkeley National Laboratory
Berkeley, USA
warndt@lbl.gov

***Abstract*—HMMER3 is biological sequence search suite used in significant volume on systems hosted at the National Energy Research Scientific Computing Center. This heavy usage has revealed ways that HMMER3 underutilizes the resources available in an HPC environment such as the Manycore architecture Knights Landing processors available in the Cori supercomputer. After rigorous performance analysis it was determined that the thread architecture of HMMER3 is the most promising optimization target to increase throughput and efficiency. A refactoring effort introduced an OpenMP task based threading design, the ability to respond to imbalanced computation with work stealing, and input buffering to eliminate a large amount of redundant parsing. These efforts have been implemented and in production on Cori for over a year. In that time they have simplified the best practice for use of HMMER3 in workflows and conserved hundreds of thousands of CPU hours.**

Keywords—Multithreading; Multicore Processing; OpenMP; High Performance Computing; Bioinformatics

I. Introduction

HMMER3 [1] is a bioinformatics application used to search a protein sequence database for contents which are statistically similar to a profile Hidden Markov Model (HMM) [2]. This application is heavily used on National Energy Research Scientific Computing Center (NERSC) systems by users from the Joint Genome Institute (JGI) for purposes such as DNA sequencing quality assurance, as a component in workflows that automatically annotate newly sequenced genomes, and novel research projects. JGI uses approximately 7 million CPU hours annually to run HMMER3 on NERSC systems.

HMMER3 is heavily optimized for the personal computing hardware of ten years ago. Data movement is organized such that very low working set memory usage has been achieved at the cost of increased file access. The application is arranged as a pipeline of filters using SSE intrinsic vector instructions to implement dynamic programming (DP) with a complex striping pattern [3]. Threading support uses pthreads via a master to worker queue dispatch that distributes sequence data to individual threads for processing. An MPI implementation is also provided in the HMMER3 distribution that does not support threading and decomposes data with the same pattern and performance as the pthread implementation.

A significant literature exists discussing the optimization of the HMMER lineage of applications. One common theme is a focus on porting low-level filter kernels to less standard platforms such as GPU accelerators [4, 5, 6], FPGA accelerators [7, 8, 9, 10], and the Cell processor architecture [11, 12]. Another pattern is adaptation of HMMER to better utilize more sophisticated HPC support systems such as better-organized network communication via MPI [13] and parallel file systems [14].

In distinction, this work describes a case study modifying HMMER3 to improve performance at a production facility with established user base, workload characteristics, hardware availability, and operational support systems. Priorities extend beyond generic speed benchmarks to include efficient utilization of allocated resources, compatibility with specific hardware and systems, consideration for labor needed to implement changes, user demanded invariance with standard HMMER3 results, ease of adoption by user base, and simplicity of integration into existing workflows and pipelines.

This project bolsters the list of example OpenMP tasking systems. Working, useful, performant, and visible (able to be found with a reasonable Google search) code using task directives is a rarity. Existing examples tend towards two types: Either a toy implementation of the Fibonacci sequence using nested tasks to perform the recursion, or walking through a linked data structure. The code implemented for this project demonstrates a useful and succinct example of OpenMP tasking achieving goals that are difficult in other threading patterns such as overlapping independent compute blocks with I/O and balancing load between units of work with divergent or unpredictable computational demands.

II. Background

A. The HMMER3 Algorithm

HMMER3 uses DP to solve a sequence alignment problem in similar manner to the Smith-Waterman algorithm. Profile HMMs are constructed to describe noteworthy protein sequence features, and candidate sequences are searched

978-1-5386-5555-9/18/$31.00 ©2018 IEEE
DOI 10.1109/IPDPSW.2018.00048

against these models up to the scale of millions of sequences and tens of thousands of HMMs.

The HMMER3 core pipeline is composed of a series of filters that discard non-matching searches quickly and cheaply. This behavior minimizes time spent in the expensive full precision DP table calculations. Initially, each sequence is processed with the Single Segment Viterbi (SSV) filter, which only considers high scoring diagonals in the DP table at 8-bit precision; a similar Multiple Segment Viterbi (MSV) filter allowing mismatches is sometimes included when SSV scores are near the threshold. If that SSV/MSV score exceeds a statistical boundary then the sequence will be passed to the more complex Viterbi filter. The Viterbi filter considers a wider range of possible sequence perturbations such as insertions and deletions, and uses 16-bit precision. A sequence passing the Viterbi filter is sent to the Forward-Backward algorithm. This final step builds and traverses the full DP table in 32-bit precision and sums score over multiple alignment paths to generate the final output.

The layer of abstraction which separates the HMMER3 core pipeline from it's top-level drivers is very well designed and facilitates a wide flexibility of parallelization. A single pipeline call requires five data structures: a protein sequence which will not be modified and can be shared with other pipelines, a HMM which can also be shared, read-only and sharable storage of background residue probabilities, a private pipeline structure which stores reusable working set memory, and a top-hits object which gathers search results for a single HMM and must only be accessed by one pipeline at a time. A parallel driver for a HMMER3 tool need only concern itself with accepting command line parameters, performing I/O, supplying these five data structures to core pipeline calls with appropriate synchronization, and organizing pipeline calls to guide search coverage of the entire input space.

B. The Cori supercomputer

Cori is the primary production machine operated by NERSC; it is ranked as the #8 fastest supercomputer worldwide as of March 2018. Cori is a Cray XC40 machine containing 2,388 32-core Xeon E5-2698 (Haswell) nodes with 128GB RAM and 9,688 68-core Xeon Phi 7250 (Knights Landing or KNL) nodes with 96GB RAM. Cori uses a Cray Aries interconnect with dragonfly topology, a Cray Sonexion 2000 Lustre file system, and includes 288 DataWarp nodes that mount high speed and high capacity SSD storage inside the Aries network. A Slurm workload manager controls resource allocation and job scheduling on Cori.

The unit of economic account on Cori is an allocation hour, corresponding to the use of one compute node for one hour scaled by factors including total job size, relative performance of node types and platforms, or a prorate by cores when a job is run using the shared queue. NERSC services support over 6,000 users leading to wide fluctuations in utilization and availability of system resources. The wait times and throughput of the various job queues are the main expression of this demand variation. A user not limited to small shared jobs but also able to effectively run `hmmsearch` on the full 32 or 68 cores of a node can more flexibly request a job submission that minimizes time until completion or obtains the most science possible from each precious allocation hour.

C. Usage of HMMER3 at NERSC

HMMER3 includes a number of applications related to the creation, manipulation, and searching of profile HMMs. High volume use of HMMER3 at NERSC is exclusively in the form of `hmmsearch`, which searches all pairs of model against protein via an outer loop over HMMs and an inner loop over sequences. For this reason, the work described here focuses only on analysis and optimization of `hmmsearch`. Use of HMMER3 on NERSC systems most resembles a High Throughput Computing model: The pairwise searching of a very large number of sequences against a large number of HMMs in `hmmsearch` is trivial to decompose with effectively no interdependencies. The active memory needed by the core pipeline for a single pair search is minuscule[1] relative to the RAM available on each node, even when permitting hundreds of threads.

The Integrated Microbial Genomes and Metagenomes database (IMG) [15] hosted by JGI is a common source of protein sequence data for HMMER3 usage at NERSC. At the time of writing, the IMG database contains 45,865,548,268 candidate protein sequences extracted from metagenome data sets. A full `hmmsearch` of IMG against the Pfam HMM database [16] using a naïve HMMER3 configuration would require more than 600,000 Haswell node hours to complete.

The needs, demands, and behaviors of NERSC users, and available systems, preclude the use of existing research on the optimization of HMMER3. There is no user demand at NERSC for any MPI implementation of `hmmsearch`; waiting a few hours for one or more single node jobs with better throughput is chosen in lieu of the user perceived mental overhead of dealing with MPI. This rules out the use of optimization projects such as MPI-HMMER [14] that target parallel file systems via an MPI layer. It further hinders MPI-HMMER that it is a port of the less powerful HMMER2 algorithm and it's source and documentation website is dead. NERSC possesses no production scale GPU or FPGA accelerated nodes so those categories of existing research are also not applicable.

Performance of Manycore architecture KNL processors depends dominantly on the need for very efficient thread implementations. This is a gap in HMMER3 optimization literature as it currently stands; a highly optimized thread implementation would be most appropriate for NERSC users and systems but has not been a development of preceding papers. The most direct comparison available to this project, and what the users originally used, is the baseline parallel code provided with the HMMER3.1b2 release.

III. Initial Performance Evaluation

A. Thread Scaling

A first experiment was conducted to measure thread scaling of baseline HMMER3.1b2 `hmmsearch`. One hundred HMMs were sampled from the Pfam 31.0 database and 100,000 sequences from the UniProt/Swiss-Prot [17] database to create

[1] Low RAM usage does not hold for a tiny number of the longest biologically valid sequences and domains. The most extreme demonstration of this would be searching a 35,991 residue TITIN protein against it's own domain.

an input file pair; ten pairs in total were created. Each input pair was passed to `hmmsearch` for execution on a Haswell node of Cori. Additional runs used the same input while progressively increasing the number of threads from 1 to 16. The strong scaling speedups determined by this experiment are presented in Fig. 1. Average wall time of a one-thread job was used to scale the speedup of each replicate; these single thread times ranged from 35.7 to 126.1 seconds with the ten replicate average being 69.4 seconds.

Results show naïve use of `hmmsearch` obtains performance benefit only from the first four to five threads and any additional have minimal positive impact. Though not shown, this trend continues with consistently flat speedup and even degradation when utilizing more than 16 threads on a Haswell node. The same scaling pattern appears when running on a KNL node but with significantly longer wall times due to lack of L3 cache and the core-per-core weakness of KNL relative to Haswell.

It is well established in the literature that the length of sequences and HMMs given to `hmmsearch` affects performance and scaling [1, 5, 6, 7, 12]. For all experiments conducted in this work a large number of sampled HMMs (100 or more) and sequences (100,000 or more) are used for experiments, along with replications of unique samples (usually 10); these large samplings are intended to reasonably emulate the distribution of input sequence and model lengths as they would occur during productive use.

B. Inconvenience as Best Practice

Given the poor performance of only using threads to scale HMMER3, many users have adapted a mitigation strategy that reclaims modest performance at the expense of an obnoxious but endurable amount of added complexity. This method uses the file system as an additional layer of parallel decomposition: split input files into shards, run multiple `hmmsearch` processes simultaneously, and then join the outputs. This idea is similar to methods devised in earlier works, though implemented purely with file manipulation instead of MPI, parallel file system, or source code modification.

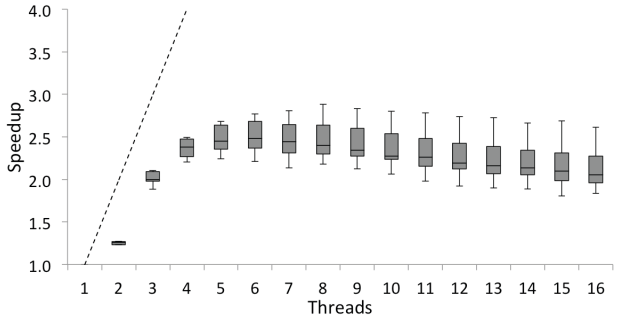

Fig. 1. Speedup achieved by thread scaling `hmmsearch` on one Cori Haswell node. Whisker plots show the aggregate speedup factor of 10 different input combinations taken from Swiss-Prot and Pfam databases. The dotted line shows the theoretical perfect scaling trendline and is intentionally cut off to preserve detail in the whisker plot.

Using the file system to parallelize `hmmsearch` is not trivial to implement on Cori. Slurm support for concurrent background execution of multiple processes on a single reserved compute node is currently not correct, so a workaround must be used which employs Multiple Program Multiple Data (MPMD) mode. Designed to connect multiple, unique, and concurrent programs to the same MPI communicator, MPMD allows the assignment of multiple process executions with unique command line parameters to disjoint sets of cores on a node. In this case the user simply ignores the MPI support.

Fig. 2 shows an example job submission script to execute an MPMD `hmmsearch` with multiple shards of an input file. A number of details in this configuration are notable. The `srun` flags `-n` and `-c` determine the total number of processes and the number of cores allocated to each, but notice their product is 64 and not the expected 32. This is because Haswell processors possess Hyper-Threading (HT) that, for resource allocation purposes, is treated as two logical cores per single physical core. Those two logical cores compete for shared resources to the extreme degree that using HT results in only a 5-10% performance gain with most applications; for the remainder of this writing HT will be avoided as not worth the trouble. It's unlikely the file split will perfectly balance load so the `-k` flag is used to disable the default MPMD behavior of ending a job when any one of its processes first exits. Note the ability to use the `%t` symbol in an MPMD configuration script to access the unique index of each task for use in parameters. Finally, outside of the scripting, there is an additional complexity for any pipeline or workflow using sharded input file `hmmsearch`, as it must assume the responsibility to divide inputs, store intermediate files, and merge output files.

An experiment was performed to demonstrate the possible configurations and performance consequences of using `hmmsearch` with split input files. The ten input file pairs used in the thread scaling experiment were reused, with the modification that sets of divided input sequence files were created to distribute their total content between 2, 4, 8, 16, and 32 files. The number of input sequence files determined the number of `hmmsearch` processes used and the number of threads allocated to each that would fully utilize 32 Haswell cores. Fig. 3 shows the speedup achieved by various ratios of threads to split files along with a theoretical ceiling based on perfect scaling of single thread performance to the full node. Amusingly, ignoring the `hmmsearch` thread implementation completely and *only* using the file system to parallelize achieves the best performance.

Slurm Batch Script

```
#SBATCH -N 1
#SBATCH -t 00:30:00
#SBATCH -C haswell
srun -n 16 -c 4 --cpu_bind=cores --multi-prog -k
  mpmd_16.conf
```

mpmd_16.conf

```
0-15 ./hmmsearch --cpu 1 -o out%t.txt
  pfam/input.%t.hmm sequence.fasta
```

Fig. 2. Example scripts to create a Slurm job which runs a sharded `hmmsearch` using 16 input files and 2 threads each

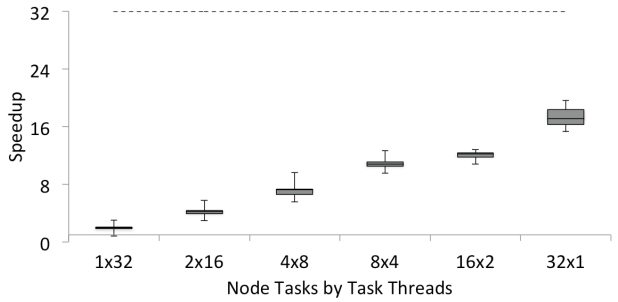

Fig. 3. Cori Haswell node `hmmsearch` speedup achieved by combining thread scaling with split input file scaling. A single whisker column shows the aggregate speedup factor of 10 different input combinations taken from UniProtKB/Swiss-Prot and Pfam databases. Six different combinations of input shard number and threads per process are shown. The dotted line shows a ceiling calculated from theoretical perfect scaling.

Note that splitting input file contents by round robin assignment of an entire sequence or HMM to each shard is not the most equal distribution of work available. Sequence length and HMM size are among factors determining needed run time so a method which distributes shard content balancing amino acid residues or model positions would reduce load imbalance between processes. The load imbalance able to be reclaimed by implementing this strategy relative to the labor needed to devise, debug, and incorporate it into a workflow is poor. I have yet to encounter an end user employing this method and it will not be considered further.

C. Performance Analysis

It has been accepted among HMMER experts that application performance in the parallel context is I/O bound. This claim was tested on Cori by staging all input files in on-node RAM, completely bypassing the file system, and measuring the wall time of a large search. Results indicated performance did not change in any measurable manner relative to use of the file system. Though at a point in the past the HMMER3 performance bottleneck may have indeed been disk access, when exposed to our usage on Cori it is not.

A second experiment was performed to determine if memory bandwidth is a factor that limits performance. It is an option on Cori to reduce the CPU clock frequency of a processor while leaving all other node systems unchanged. If the clock speed is reduced 50% and the run time doubles then the application is compute bound. If the run time increases by less than double, such as when less frequent access can be served by the memory system with fewer stalls, then it suggests memory bandwidth or latency is indeed a limit on performance. Experiments running `hmmsearch` with modified clock speed suggest less than 4% of the running time can be attributed to memory performance limitations.

The thread implementation of `hmmsearch` poses a curious difficulty for all modern performance analysis tools. Each input HMM causes the master thread to fork a number of child threads equal to the `--cpu` flag. When computation is complete these threads are discarded, the next HMM is read, and a new set of threads are forked. Threading analysis tools shown this behavior cannot track the relationship between subsequent groups of forked threads and instead report thousands of independent and temporally disjoint threads each with a tiny fraction of the total compute. Any rigorous analysis of `hmmsearch` thread performance must employ a manual aggregation of this information into comprehensible form.

The CrayPat performance analysis tool was used to collect performance data while running `hmmsearch`. The executable was augmented with `pat_build` to collect function call sampling data. Functions were sorted into four categories: input sequence I/O and parsing, thread creation and destruction, thread blocking, and computation. Large sampling reports of thread behavior were manually manipulated in a spreadsheet to aggregate behavior by category and distinguish master and worker threads.

Understanding these results requires a detailed explanation of the threading model and data structure used to parallelize `hmmsearch`. Threads are organized around a single master that reads and parses all input, maintains a synchronized queue and loads units of work into it, spawns and destroys worker threads, removes completed work from the queue, and writes output to file. A team of worker threads is created for each input HMM and the synchronized queue distributes sequences to the workers in this team for search against the HMM. When all searches against that HMM are complete the worker team is destroyed. The process is repeated until all HMMs have been searched.

Function sampling results presented in Table I. demonstrate the full range of pathology when `hmmsearch` master and worker threads distribute work amongst themselves.

Choosing no worker threads activates the serial version of `hmmsearch` and incurs no thread overhead or load balance problem; the ratio of I/O to compute is 1:6, which is conspicuously close to the empirically determined ideal number of worker threads.

When one worker thread is present the master thread performs essentially the same amount of I/O work as it does in the serial case, but fills all time previously used for compute with thread blocking, yielding a very similar total wall time. The unimpressive speed gain can be explained as the master thread filling the work queue faster than one worker can empty it. A maximum queue capacity is quickly reached where the master blocks as it waits for additional input space to become available.

TABLE I. FUNCTION SAMPLING OF HMMSEARCH FOR VARYING NUMBER OF WORKER THREADS.

	Master			Average Worker		
Work Thread	*I/O*	*Join*	*Wait*	*Compute*	*Wait*	*Time (seconds)*
0	14%			83%		127
1	17%		81%	99%		114
3	41%	20%	37%	71%	28%	54
7	50%	32%	15%	44%	55%	47
15	46%	41%	9%	27%	72%	51

An experiment with fifteen worker threads demonstrates in extremis the opposite imbalance relative to one fully loaded worker thread. In this second case, the master thread has increased it's fraction of time spent performing I/O by a factor of three and significantly reduced the amount of queue related blocking, but replaced that queue spinning with thread fork and join overhead. The average worker thread spends only 25% of execution performing search while the rest is lost blocking on the synchronized queue as it waits for new data to become available. This behavior is a result of the single master thread being unable to supply sequence at a rate comparable to the rate at which workers consume it. Additionally, the master is burdened by thread creation and destruction overhead. This further explains why split input file `hmmsearch` scales so much more effectively: having more than one master thread in different processes both reduces thread overhead for each and enables sequence parsing to occur in parallel. The overall effect is the rate of input preparation increases and more worker threads can be effectively supplied with data.

Results from CrayPat experiments also aggregate to suggest a target for optimization. Less than 2% of the sampled time attributed to I/O is spent in system buffered blocking read calls; the rest is in `sqascii_Read()` and `header_fasta()`, both of which are input parsing functions. This glut is a direct consequence of the data access pattern in the `hmmsearch` top-level application. A small overhead is needed to read, parse, and error check one sequence from disk, but all such work is discarded and duplicated for each new model. These parsing functions are a primary factor limiting the rate new sequences can be added to the thread-dispatching queue and thus total application throughput.

IV. Modifications

A. New hpc_hmmsearch Driver

All of the following optimizations have been implemented by duplicating and then modifying only the top-level `hmmsearch` driver (`hmmsearch.c`) into a new driver (`hpc_hmmsearch.c`) while leaving the core pipeline intact. Code implementing the pthread or MPI use of the synchronized work queue was removed and replaced with a system based on OpenMP task directives. Restricting the scope of modification to only the top-level driver significantly reduced the analysis, engineering, quality assurance, and time necessary to complete the project. The new driver application will be referred to as `hpc_hmmsearch`.

B. Input Data Buffers

The first major modification was to change data access such that parsed sequence input data is retained in memory buffers. These buffers store sequences and models such that each traversal through an input file provides data to perform multiple core pipeline calls. An attractive tradeoff is introduced where using a modest amount of additional memory significantly reduces the number of times entire input files must be loaded and parsed.

With the expense of a few hundred extra megabytes of RAM, which is abundantly available on Cori hardware, the amount of CPU used to parse and error check sequence file data can be reduced by a factor of 20 or more. This eliminates 25% of the total application computation off the top before considering any other configuration decisions or optimizations.

C. Concurrent HMM searches

The distribution of HMMs amongst worker threads has been changed in `hpc_hmmsearch`. The original decomposition reads a single HMM, copies it to each worker, divides input sequences between all threads, synchronizes threads, and aggregates results when all searches against that model are complete. This arrangement minimizes the time to search a single model but introduces one thread synchronization and the potential manifestation of load imbalance for each additional model. The modified driver distributes a unique HMM to each thread such that threads do not need to sync and collectively aggregate reports for output. Individual threads completing their assigned searches can immediately output results and accept a new HMM without dependency on other threads or reducing data structures. The result is a more efficient packing of compute and fewer global synchronization points, at the expense of reduced performance when the number of HMMs is small relative to the number of CPU cores (a situation which does not fit our usage).

D. Overlapping I/O and Compute Using OpenMP Tasking

All pthread code has been replaced with an OpenMP implementation based on task directives. At the top level of behavior, `hmmsearch` input file contents define an all-against-all search area. The total area of this search can be arbitrarily subdivided and the resulting rectangles streamed to teams of worker tasks while I/O tasks execute alongside but on input required during the next rectangle of work in the stream. The change was implemented as follows:

Two pairs of buffers are created: a pair storing HMMs and a pair storing sequence data; elements of each pair are referred to as "flip" and "flop". A top-level loop contains all the tasks needed for a rectangle of work within a `taskgroup` directive and iterates until all models in the HMM file have been fully searched. I/O tasks write the HMM flop buffer to output if it contains complete results and read sequence or HMM data needed by the next `taskgroup` into flop buffers. Concurrently, worker tasks perform core pipeline searches of sequences and HMMs in the flip buffers, and deposit their results into top-hits structs in the HMM flip buffer. When all tasks in the `taskgroup` are complete the content of the sequence flip and flop buffers are exchanged. This swap moves the next rectangle of unprocessed searches to where they will be accessed by worker tasks during the next `taskgroup`. After a full scan through the sequence database file, the flip and flop HMM buffers are swapped and the sequence file is rewound.

With this organization, and reasonable configuration of buffer sizes, workers do not need to wait for disk access, parsing, or thread synchronization events. When a file access task finishes before overall processing is complete it will automatically convert itself to an additional worker thread.

E. Dynamic Load Balancing

Load balancing `hmmsearch` is a challenge due to the highly conditional nature of its control flow. Though it is simple enough to dispatch equal amounts of sequence to each worker or balance the number of residues, uneven concentrations of search hits advancing deeper into the pipeline and consuming disproportionate resources cannot be anticipated and lead to significant risk of imbalance. This effect has been modestly mitigated using OpenMP `taskgroup` to implement a work stealing mechanism in `hpc_hmmsearch`.

All worker tasks are issued within a `taskgroup` directive and the number of outstanding tasks is tracked; when that number falls below the number of cores available then tasks with a sufficient amount of unprocessed sequence will create a new child task and pass half their remaining work to it. The child task is then immediately scheduled to execute by the runtime and occupies the empty core. The `taskgroup` directive is needed to guarantee all child tasks spawned by the work stealing mechanism have finished before finalizing a rectangle of work and swapping buffers (a `taskwait` directive would only collect all tasks at the same depth before releasing, ignoring any child tasks that may remain outstanding).

V. Results

An initial performance experiment is included to directly contrast the thread-scaling difference between the original `hmmsearch` driver and `hpc_hmmsearch`. Fig. 4 presents data obtained by running both `hmmsearch` drivers on the Edison system with the same configuration as used to create Fig. 1 (Edison is architecturally analogous to Cori Haswell, but one generation older with 24 cores per node instead of 32).

Best practice performance has been evaluated by running both `hpc_hmmsearch` and sharded file `hmmsearch` with a new larger set of sampled input files on both types of Cori nodes. The entire Pfam-A database was used for each experimental search by creating sets of divisions that round-robin distribute all Pfam HMMs between 2, 4, 8, 16, 32, and 64 files. Note that this division of the HMM database is different from earlier file shard experiments which divided the sequence database (A minor experiment confirmed that either division is equivalent in terms of performance, though in practice, dividing the HMM database is preferred because it is more convenient to implement in a workflow). Sequence files for this experiment were created via sampling one million protein sequences from the UniProtKB/TrEMBL database.

Fig. 5 shows scaling performance results when running the most effective Pfam file decompositions with `hmmsearch` and `hpc_hmmsearch` on a Haswell Cori node. Reported speedup is scaled against the wall time of running one single threaded `hmmsearch` job with the same input. The best performing `hmmsearch` among the available configurations, using 32 serial processes, yields an average of 17.4 speedup. Average speedup of `hpc_hmmsearch` is 26.4, a 51% improvement and 61% closer to the theoretical maximum throughput.

Fig. 6 displays Knights Landing node scaling performance results for a range of `hmmsearch` file decompositions and `hpc_hmmsearch`. The same series of Pfam input files were used as the Haswell performance experiment, but the number of sequences sampled was reduced by 80% to 200,000. The reported speedup is scaled to the projected time needed to run one single thread `hmmsearch` on a KNL node. A dashed line marks an upper bounded speedup of 136, corresponding to the full utilization of two hardware threads on each KNL core. Of several reasonable decompositions, the best performing `hmmsearch` configuration is 32 input file shards with 4 threads each, producing an average speedup factor of 57.5. The KNL `hpc_hmmsearch` running with 136 threads yields an average speedup factor of 116.1; this is a 101% improvement above `hmmsearch` best practice on the KNL platform and almost 4x closer to the theoretical maximum. Though KNL speedup gains are much better than Haswell, the base performance of KNL is handicapped enough that the shortest wall times using the best configurations are still obtained using a Haswell node.

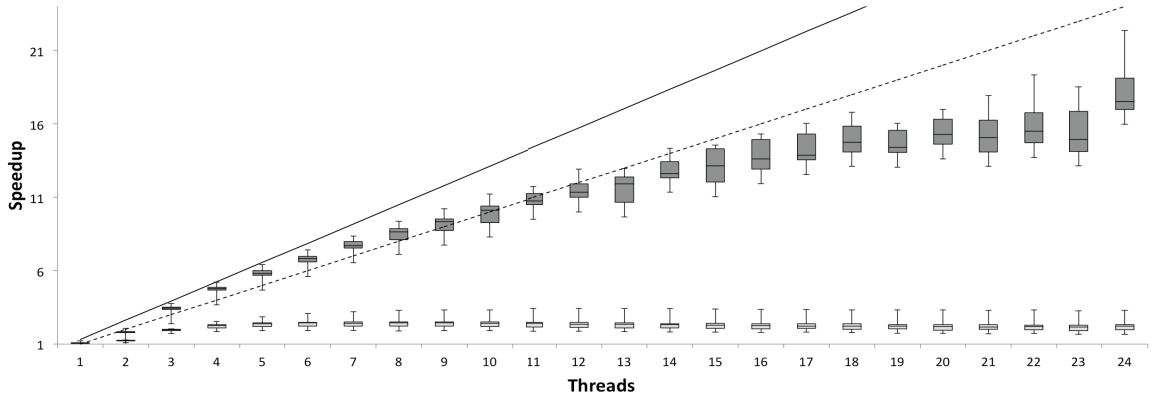

Fig. 4. Speedup achieved by thread scaling `hmmsearch` (light) and `hpc_hmmsearch` (dark) on one Edison Ivy Bridge node with 24 cores. Whisker plots show the aggregate speedup factor of 10 different input combinations taken from Swiss-Prot and Pfam databases. The dotted line shows the theoretical perfect scaling trendline of the `hmmsearch` runs. The solid line shows the theoretical perfect scaling trendline of `hpc_hmmsearch` progression.

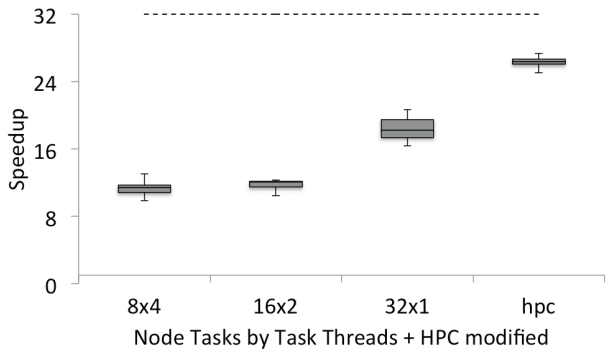

Fig. 5. Cori Haswell node `hmmsearch` speedup achieved by the three best file split regimes, and the same input run with `hpc_hmmsearch` (hpc). Whiskers show the aggregate speedup factor of 10 different input combinations taken from UniProtKB/TrEMBL and Pfam databases. Dotted line shows the theoretical perfect scaling ceiling.

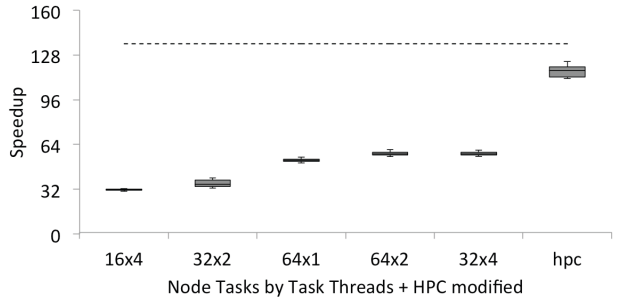

Fig. 6. Cori KNL node `hmmsearch` speedup achieved by various file split regimes, and the same input run using `hpc_hmmsearch` (hpc). Whiskers show the aggregate speedup factor of 10 different input combinations taken from UniProtKB/TrEMBL and Pfam databases. Dotted line shows a theoretical perfect scaling ceiling assuming 2 hardware threads per core.

VI. Discussion

One important point is examination of the performance discrepancy between running HMMER3 on Haswell vs. KNL nodes. Optimally configured and everything else being equal, a HMMER3 job will always run faster on a Haswell node than on a KNL node. Haswell completes a single-thread `hmmsearch` job 3.9x faster than the same single-thread job on KNL, a well-configured split file job 1.2x faster, and an `hpc_hmmsearch` job 1.45x faster. The flexibility to run either is, however, still valuable as being able to run HMMER3 well on KNL can still benefit users when balancing allocation charge factors, queue demand differences, and the fact that over four times as many KNL nodes are available on Cori.

What characteristics of these systems make Haswell more suitable to run HMMER3 than KNL? The primary suspects are the core filter kernels implemented using SSE vector instructions. Each KNL core operates at half the frequency of a Haswell and only has one vector-processing unit (VPU) able to execute legacy vector instructions. Any possible gains from the 4x wider AVX-512 vectors or the 4 hardware threads per core can't be realized through the VPU bottleneck as the HMMER3 core is currently implemented.

What consideration has been given to improving core pipeline components? The ratio of effort to reward when considering a vector modernization of HMMER3 pipeline kernels offers an intimidating calculus. Most potential upside for NERSC users is cut off at the knee because HMMER3 core components use byte and word sized instructions, but KNL silicon does not support the AVX-512BW instruction set which implements them. Upgrading software to AVX2 (256 bit vector width instructions) could theoretically offer up to 2x speedup but that would break compatibility with Edison (NERSC's previous generation production system), KNL performance would still be limited to the single legacy VPU per core, and the modifications would incur an enormous labor effort to manually convert, debug, and verify every pipeline component, data structure, and post-processing routine in the entire software stack. Furthermore, the HMMER4 development team has claimed and finished implementing support for all modern vector instruction sets; any work in that direction would be redundant. This does not only apply to vectorization porting; *all* work on HMMER3 will become obsolete when HMMER4 is released. To be successful this project needed to be completed and enter production well before then.

A revised calculation of the estimated time needed to search IMG against Pfam quantifies the benefit to NERSC users since `hpc_hmmsearch` has entered production. I must concede the initial 600,000 CPU hour estimate was incendiary; a real power user would not use the naïve decomposition but instead split Pfam into 8 parts, IMG into 2,500 fasta files, and submit 2,500 jobs each packing 8 `hmmsearch` with 4 threads. The total allocation would use approximately 60,000 CPU hours, be throttled by the policy limit of 200 queued jobs per user, and take 25 days to pass entirely through the queue. A second researcher generating the same data using `hpc_hmmsearch` could split 1,000 fasta files, queue 1,000 jobs, consume 23,000 CPU hours, and fully clear the queue in 10 days.

Collaboration with a FICUS JGI-NERSC [18] project provides a detailed real world anecdote of the performance gained using `hpc_hmmsearch`. A component in the project workflow required a HMMER3 search of 2.2 billion IMG protein sequences against 229 curated HMMs. Basic use of `hmmsearch` for this task is estimated to be almost 60,000 CPU hours while split file configuration would consume approximately 7,500 hours. These usage numbers are overestimates because they are calculated with scaling factors derived from our earlier experiments; the distribution of the project's metagenomic data contains fewer search hits and thus spends a smaller fraction of time in deeper pipeline stages. We worked with the project to integrate `hpc_hmmsearch` into their workflow. The resulting work split the input sequence into 184 parts of 12 million sequences each and consumed only 130 CPU hours.

The pattern of OpenMP tasking demonstrated in this project is applicable and beneficial to a wider and more general set of applications appearing in the bioinformatics toolbox.

Such applications follow a pattern of reusing one or more well-abstracted core kernels that do not induce side effects or data dependencies on other kernel calls. In these cases some tasks can perform I/O and prepare future data buffers while other tasks issue the kernel calls on prepared data or empty completed buffers to output. Throughput limited reductions, local filters, or all-against-all algorithms such as string or similarity searches can directly mimic this design. The pattern can also accommodate methods that require large and shared read-only data structures such as an FM-index or k-mer table. All worker tasks can trivially share a single copy of that data structure in the OpenMP shared memory environment.

VII. Conclusion

This work describes in detail the process of adapting HMMER3 to better conform to the needs of the NERSC user community and the Cori supercomputer. Performance analysis indicated that data processing in the top level driver of `hmmsearch` was the best target for optimization efforts. Only a modest amount of labor was necessary to refactor the threading architecture to use OpenMP tasking, overlap I/O and computation activity, automatically load balance between tasks, reduce I/O overhead by buffering input sequence, and simplify the best practice use of `hmmsearch` in a workflow context. The h`pc_hmmsearch` driver has been in production on NERSC systems for over a year at the time of this publication and has cumulatively reduced system-load by hundreds of thousands of CPU hours.

Source Code

The `hpc_hmmsearch` driver can be obtained at https://github.com/Larofeticus/hpc_hmmsearch along with instructions for installation and usage.

Acknowledgment

The research and development described in this writing has been generously supported by the Joint Genome Institute, the Intel Parallel Computing Center at Lawrence Berkeley National Laboratory, and the NERSC Exascale Science Applications Program.

References

[1] S. Eddy, "Accelerated profile HMM searches," *PLoS Computational Biology*, vol. 7, no. 10, 2011. doi:10.1371/journal.pcbi.1002195

[2] S. Eddy, "Profile hidden markov models," *Bioinformatics*, vol. 14, pp. 755-763, 1998.

[3] M. Farrar, "Striped Smith-Waterman speeds database searches six times over other SIMD implementations," *Bioinformatics*, vol. 23, pp. 156-161, 2007.

[4] D. Horn, M. Houston, and P. Hanrahan, "ClawHMMER: A streaming HMMer-search implementation," in *Proceedings of ACM/IEEE Supercomputing Conference*, 2005.

[5] X. Li, W. Han, G. Liu, H. An, M. Xu, W. Zhou, and Q. Li, "A speculative HMMER search implementation on GPU," in *IEEE 26th IPDPS Workshop and PhD Forum*, 2012, pp. 73-74.

[6] H. Jiang and N. Ganesan, "Fine-grained acceleration of hmmer 3.0 via architecture-aware optimization on massively parallel processors," in *Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International*. IEEE, 2015, pp. 375-383.

[7] T. Oliver, L. Y. Yeow and B. Schmidt, "High Performance Database Searching with HMMer on FPGAs," *Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1-7. doi:10.1109/IPDPS.2007.370448

[8] S. Derrien and P. Quinton, "Parallelizing HMMER for Hardware Acceleration on FPGAs," in *IEEE ASAP*, 2007, pp. 10-17.

[9] Y. Sun, P. Li, G. Gu, Y. Wen, Y. Liu and D. Liu, "Accelerating HMMer on FPGAs Using Systolic Array Based Architecture," in *IEEE IPDPS*, 2009, pp. 1-8.

[10] T. Takagi and T. Maruyama, "Accelerating HMMER Search Using FPGA," in *Proceedings of the 19th International Conference on Field-Programmable Logic and Applications*, No. T3C1, 2009.

[11] J. Lu, M. Perrone, K. Albayraktaroglu, and M. Franklin. "HMMer-Cell : High Performance Protein Profile Searching on the Cell/B.E. Processor." *IEEE International Symposium on Performance Analysis of Systems and software*, 2008, pp. 223-232.

[12] S. Isaza, E. Houtgast and G. Gaydadjiev, "HMMER Performance Model for Multicore Architectures," *14th Euromicro Conference on Digital System Design*, Oulu, 2011, pp. 257-261. doi: 10.1109/DSD.2011.111

[13] J. P. Walters, B. Qudah and V. Chaudhary, "Accelerating the HMMER sequence analysis suite using conventional processors," *20th International Conference on Advanced Information Networking and Applications – Vol. 1*, 2006, pp. 6. doi: 10.1109/AINA.2006.68

[14] J. P. Walters, R. Darole and V. Chaudhary, "Improving MPI-HMMER's scalability with parallel I/O," *2009 IEEE International Symposium on Parallel & Distributed Processing*, Rome, 2009, pp. 1-11. doi: 10.1109/IPDPS.2009.5161074

[15] V. M. Markowitz, I. A. Chen, K. Palaniappan, K. Chu, E. Szeto, Y. Grechkin, and A. Ratner. "IMG: the Integrated Microbial Genomes database and comparative analysis system." *Nucleic Acids Research*, Database Issue 40, 2012, D115-D122.

[16] R.D. Finn, P. Coggill, R.Y. Eberhardt, S.R. Eddy, J. Mistry, and A.L. Mitchell. "The Pfam protein families database: towards a more sustainable future." *Nucleic Acids Research*, Database Issue 44, 2016, D279-D285.

[17] The UniProt Consortium. "UniProt: the universal protein knowledgebase." *Nucleic Acids Research*, Database Issue 45, 2017, D158-D169.

[18] https://jgi.doe.gov/user-program-info/community-science-program/how-to-propose-a-csp-project/ficus-jgi-nersc/