PhyloBayes MPI

Harry Waugh

***Abstract*—This report surveys different Intel and ARM processors to understand which systems can be exploited to maximize the performance of the PhyloBayes MPI code [1]. The report identifies a key bottleneck in the `Propagate` function which can be significantly reduced by vectorizing various loops using the Intel compiler. Analysis is also given on where effort should be focused in the future to further improve performance.**

I. Identifying the bottlenecks in PhyloBayes

To gain an understanding of where the bottlenecks were in the PhyloBayes code, the TAU performance system was used to profile and gather a trace. The trace is shown in Figure 1 and revealed the functions that were responsible for the majority of the run-time. A key bottleneck was discovered to be the `Propagate` function within the `MatrixSubstitutionProcess` class, which contained 3 loops that dominated the run-time.

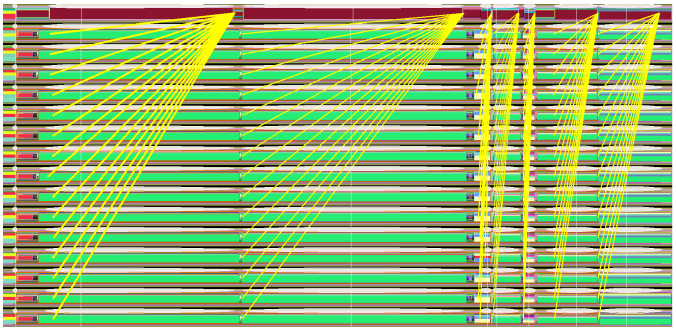

Figure 1: A TAU trace of the PhyloBayes MPI code with 16 processes. This identified a major bottleneck of the code to be the `Propagate` function (green blocks).

II. Vectorizing PhyloBayes

Section I noted that the performance of the PhyloBayes program was limited by 3 simple loops. None of these featured aliasing pointers, and each loop was an ideal candidate for vectorization. To determine whether the compiler was currently vectorizing the code, vectorization reports were generated for both the GNU and Intel compilers. From these, it was found that neither compiler vectorized all 3 of the loops when compiled with the `-O3` vectorization flag.

Reports were then generated again after adding flags that specified the processor's instruction set, which was AVX2 for an Intel E5-2680 v4. These AVX2 instructions utilize 256 bit registers to operate on 4 double precision floating point numbers with a single instruction. The vectorization reports showed that the Intel 2018 compiler was successful at vectorizing each loop, whereas the GNU compiler failed to vectorize one on the loops. The consequence of this failed vectorization is reflected in Table II, which shows the Intel 2018 compiler providing a 2.3x speedup over the GNU compiler. The Intel 2017 compiler was 1.14x slower than the 2018 compiler, as one of the `Propagate` loops did not vectorize.

Analyzing the vectorization reports revealed that the `exp()` function was preventing the GNU compiler from vectorizing one of the loops. GCC 7.2 and glibc 2.17 don't support the vectorization of the `exp()` function, whereas the Intel 2018 compiler provides a more comprehensive vectorizable math library.

Further research was later completed to determine whether modern versions of glibc (2.3) and GCC (9.1.0) could match the proprietary software's performance. Benchmark tests were run using a consumer grade Intel i7-7700k, featuring 4 cores capable of AVX2 instructions. The results of this test are shown in Table II and show significantly better performance compared to Intel. Now that GNU vectorized the exponential function, it was only 1.26x slower than Intel. This was a dramatic improvement over past results, where it was over 2x slower. For reference, GNU release notes indicate that glibc contains the vector math library (`libmvec`) from version 2.22 or later [2].

III. Scaling on Intel Cluster

Following the successful vectorization of PhyloBayes using the Intel compiler, the code was tested on 4 different node configurations to test the program's scalability. The run-time for each configuration can be seen in Figure 2, which shows a significant decrease in run-time when increasing from 1 socket to 1 node, and 1 node to 2 nodes. Unfortunately, this performance gain isn't replicated when increasing from 2 nodes to 4 nodes, which only provides a speedup of 1.1x.

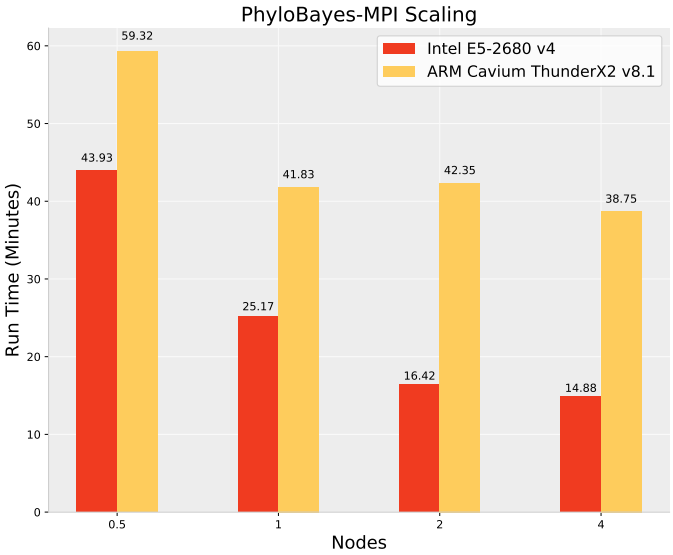

Figure 2: Benchmark times for completing 75 updates using different node configurations. Each Intel node contains 28 cores and each ARM node contains 64 cores.

IV. ARM ThunderX2

This section looks at the performance of the PhyloBayes code on an Cavium ThunderX2 ARM processor. This architecture differs with the previously tested Intel processor in two ways that are relevant for the PhyloBayes code.

The first is the smaller vector register width of 128 bits, which is half that of the Intel processor. Section II shows that this affects the cost of the `Propagate` function, as the ARM processor is restricted to operating on 2 double precision numbers at once. The second difference is that the ARM core contains 32 cores per socket, which is more than twice the 14 cores in the Intel socket.

The Cray compiler was used to compile the code on the ARM system, which ran the benchmark test in 41 minutes, as shown in Table II. The compiler successfully vectorized the 3 loops in the `Propagate` function, but still ran a significant 1.63x slower than the Intel node. The scalability of the code on the ARM processor was also tested and can be seen in Figure III. This shows that the run-time decreases by 1.41x when increasing from 1 to 2 sockets, however there was no speedup when running PhyloBayes across multiple nodes. Interestingly, both the Intel and ARM processor provide diminishing decreases in run-time beyond utilizing 56 cores.

V. Communication Pattern

This section examines the communication pattern of the PhyloBayes code, and the impact this has on the scalability of the program.

The PhyloBayes code operates a master/slave communication model, where the master broadcasts a signal which indicates the function that should be computed next. A profile was collected for similar amounts of cores on both ARM and Intel configurations to identify the proportion of time spent in the `Propagate` and `MPI_Bcast` functions, this is shown in Table I.

Table I: Proportion of time spent in the `Propagate` and `MPI_Bcast` functions, for an Intel E5-2680 a ARM ThunderX2.

	Intel		**ARM**	
Function	**1 Node (28 cores)**	**2 Nodes (56 cores)**	**1 Socket (32 cores)**	**1 Node (64 cores)**
`Propagate`	30%	22%	40%	28%
`MPI_Bcast`	27%	53%	28%	42%

This shows that the increase from 28/32 to 56/64 cores switches the primary bottleneck of PhyloBayes from the `Propagate` function to the MPI broadcast function. This is not surprising, considering the communication topology and that `MPI_Bcast` is blocking, as more processes must be ready before starting the broadcast.

A possible solution to this problem would be to incorporate a non-blocking broadcast, `MPI_Ibcast`, which would allow processes to complete some 'work' while waiting for other processes to be ready. However, this is not currently possible, as each slave process depends on the broadcast signal to indicate which work to compute next.

VI. AVX-512

Given that incorporating more cores provided diminishing returns, focus was switched back to vectorization, which enabled parallelism without the communication overheads. The code was tested with a Skylake Intel Gold 5120, which makes use of larger 512-bit registers using the AVX-512 instruction set. The results shown in Table II were compiled with the flags, `-xCORE-AVX512` and `-qopt-zmm-usage=high`, to enforce vectorisation.

Unfortunately, the AVX-512 instructions did not provide any speedup over AVX2 and actually caused an increase in run-time. Currently, online sources and rudimentary tests indicate that the reason for this behaviour is due to the CPU dynamically throttling down the clock speed to handle the AVX-512 instructions. [3].

VII. Conclusion and Future Work

After comparing PhyloBayes' performance on a range of Intel and ARM system configurations, it is clear that the best performance is achieved with processors that support the AVX2 instruction set. These processors achieve a good degree of parallelism and incur less communication overheads that their ARM counterparts when broadcasting information.

To further improve the performance of PhyloBayes, this report recommends 3 future avenues of research:

- Section II detailed how vectorization significantly improved the Intel compiler's performance when compared to GNU. Effort should therefore be put in to understand why AVX-512 does not further improve performance. Alternatively, the use of single floating point numbers would also allow more numbers to be operated on per instruction.
- Section V shows how broadcasting becomes a major bottleneck when using ARM architectures that contain a large number of cores. Thus, research should be done to determine if independent work can be completed while waiting for a non-blocking broadcast to complete, and whether this will help mask the communication overheads.

Table II: Benchmark results for PhyloBayes MPI.

Hardware (cores)	**Compiler**	**Time (minutes)**
2 Intel E5-2680 (28)	GCC 7.2	57.52
2 Intel E5-2680 (28)	Intel 2017-u1	28.77
2 Intel E5-2680 (28)	Intel 2018-u3	25.17
1 Intel i7-7700k (4)	Intel 2019-u1	111.46
1 Intel i7-7700k (4)	GCC 9.1.0	140.34
2 ThunderX2 ARM v8.1 (64)	Cray 8.7.9	41.83
2 Intel Gold 5120 (28)	Intel 2018-u3	34.56

References

[1] "PhyloBayes-Github." https://github.com/bayesiancook/pbmpi.

[2] "Glibc 2.22 release notes." https://www.sourceware.org/ml/libc-alpha/2015-08/msg00609.html.

[3] "Variable Clock Speeds." https://www.tcm.phy.cam.ac.uk/~mjr/IT/clocks.html.