Enhancing Application Performance using Mini-Apps: Comparison of Hybrid Parallel Programming Paradigms

Gary Lawson
Old Dominion University
1300 Engineering & Computational Sciences Building
Norfolk, VA, USA 23529
glaws003@odu.edu

Masha Sosonkina
Old Dominion University
1300 Engineering & Computational Sciences Building
Norfolk, VA, USA 23529
msosonki@odu.edu

Robert Baurle
NASA Langley Research Center
1 Nasa Dr.
Hampton, VA, USA 23666
robert.a.baurle@nasa.gov

Dana Hammond
NASA Langley Research Center
1 Nasa Dr.
Hampton, VA, USA 23666
dana.p.hammond@nasa.gov

ABSTRACT

In many fields, real-world applications for High Performance Computing have already been developed. For these applications to stay up-to-date, new parallel strategies must be explored to yield the best performance; however, restructuring or modifying a real-world application may be daunting depending on the size of the code. In this case, a mini-app may be employed to quickly explore such options without modifying the entire code. In this work, several mini-apps have been created to enhance a real-world application performance, namely the VULCAN code for complex flow analysis developed at the NASA Langley Research Center. These mini-apps explore hybrid parallel programming paradigms with Message Passing Interface (MPI) for distributed memory access and either *Shared MPI* (SMPI) or *OpenMP* for shared memory accesses. Performance testing shows that MPI+SMPI yields the best execution performance, while requiring the largest number of code changes. A maximum speedup of 23× was measured for MPI+SMPI, but only 11× was measured for MPI+OpenMP.

KEYWORDS

Mini-apps, Performance, VULCAN, Shared Memory, MPI, OpenMP

ACM Reference format:
Gary Lawson, Masha Sosonkina, Robert Baurle, and Dana Hammond. 2017. Enhancing Application Performance using Mini-Apps:
Comparison of Hybrid Parallel Programming Paradigms. In *Proceedings of ACM EuroMPI/USA conference, Chicago, IL USA, September 2017 (EuroMPI/USA'17),* 5 pages.
DOI:

1 INTRODUCTION

In many fields, real-world applications have already been developed. For established applications to stay up-to-date, new parallel strategies must be explored to determine which may yield the best performance, especially with advances in computing hardware. However, restructuring or modifying a real-world application incurs increased cost depending on the size of the code and changes to be made. A mini-app may be created to quickly explore such options without modifying the entire code. Mini-apps reduce the overhead of applying new strategies, thus various strategies may be implemented and compared. This work presents the authors experiences when following this strategy for a real-world application developed by NASA.

VULCAN (Viscous Upwind Algorithm for Complex Flow Analysis) is a turbulent, nonequilibrium, finite-rate chemical kinetics, Navier-Stokes flow solver for structured, cell-centered, multiblock grids that is maintained and distributed by the Hypersonic Air Breathing Propulsion Branch of the NASA Langley Research Center [13]. The mini-app developed in this work uses the Householder Reflector kernel for solving systems of linear equations. This kernel is used often by different workloads, and is a good candidate to decide what strategy type to apply to VULCAN. VULCAN is built on a single-layer of MPI and the code has been optimized to obtain perfect vectorization, therefore two-levels of parallelism are currently used. This work investigates two flavors of shared-memory parallelism, OpenMP and Shared MPI, which will provide the third-level of parallelism for the application. A third-level of parallelism increases performance, which decreases the time-to-solution.

MPI has extended the standard to MPI version 3.0, which includes the Shared Memory (SHM) model [11, 12], known in this work as Shared MPI (SMPI). This extension allows MPI to create memory windows that are shared between MPI tasks on the same physical node. In this way, MPI tasks are equivalent to threads, except Shared MPI is more difficult for a programmer to implement. OpenMP is the most common shared-memory library used to date because of its ease-of-use [14]. In most cases, only a few OpenMP pragmas are required to parallelize a loop; however, OpenMP is subject to increased overhead, which may decrease performance if not properly tuned.

The major contributions of this work are as follows:

- Created mini-apps to solve $AX = B$ using the Householder reflector kernel from NASA VULCAN real-world code
- Applied MPI+OpenMP scheme to create the OpenMP mini-app
- Applied MPI+SMPI scheme to create the Shared MPI mini-app
- Validated numerical output of each mini-app
- Compared execution performance of all mini-apps

1.1 Related Work

As early as the year 2000, the authors in [1] found that latency sensitive codes seem to benefit from pure MPI implementations whereas bandwidth sensitive codes benefit from hybrid MPI+OpenMP. Also, the authors found that faster processors will benefit hybrid MPI+OpenMP codes if data movement is not an overwhelming bottleneck [1].

Since this time, hybrid MPI+OpenMP implementations have improved, but not without difficulties. In [2, 3], it was found that OpenMP incurs many performance reductions, including: overhead (fork/join, atomics, etc), false sharing, imbalanced message passing, and a sensitivity to processor mapping. However, OpenMP overhead may be hidden when using more threads. In [15], the authors found that simply using OpenMP could incur performance penalties because the compiler avoids optimizing OpenMP loops – verified up to version 10.1. Although compilers have advanced considerably since this time, application users that still compile using older versions may be at risk if using OpenMP. In [2, 3] the authors found that the hybrid MPI+OpenMP approach outperforms the pure MPI approach because the hybrid strategy diversifies the path to parallel execution.

More recently, MPI extended its standard to include the SHM model [12]. The authors in [9] present MPI RMA theory and examples, which are the basis of the SHM model. In [5], the authors conduct a thorough performance evaluation of MPI RMA, including an investigation of different synchronization techniques for memory windows. In [8], the authors investigate the viability of MPI+SMPI execution, as well as compare it to MPI+OpenMP execution. It was found that an underlying limitation of OpenMP is the shared-by-default model for memory, which does not couple well with MPI since the memory model is private-by-default. For this reason, MPI+SMPI codes are expected to perform better, since shared memory is explicit and the memory model for the entire code is private-by-default.

Most recently, a new MPI communication model has been introduced in [6], which better captures multinode communication performance, and offers an open-source benchmarking tool to capture the model parameters for a given system. Independent of the shared memory layer, MPI is the *de facto* standard in data movement between nodes and such a model can help any MPI program. The remainder of this paper is organized into the following sections: 2 introduces the Householder mini-apps, 3 presents the performance testing results for the mini-apps considered, and 4 concludes this paper.

2 HOUSEHOLDER MINI-APP

The mini-apps use the householder computation kernel from VULCAN, which is used in solving systems of linear equations. The householder routine is an algorithm that is used to transform a square matrix into triangular form, without increasing the magnitude of each element significantly [7]. The Householder routine is numerically stable, in that it does not lose a significant amount of accuracy due to very small or very large intermediate values used in the computation.

The routine works through an iterative process of utilizing Householder transformations to annihilate elements from the column-vectors of the input matrix. The Householder reflector H is applied to a system as:

$$(HA)x = Hb, \qquad \text{where} \qquad H = (I - 2vv^T)a_i. \qquad (1)$$

The Householder operates on a_i, which is a column of A, and v, which is a unit-vector perpendicular to the plane by which the transform is applied. A more detailed discussion of the Householder routine can be found in [7]. In this work, the problem to be solved is $AX = B$, where A is a 3-dimensional matrix of size $m \times n \times n$, and X and B are 2-dimensional matrices of size $m \times n$. Each system, represented by m, is independent of all other systems; therefore, this algorithm is embarrassingly parallel.

2.1 Mini-App Design

Mini-apps are designed to perform specific functions. In this work, the important features are as follows:

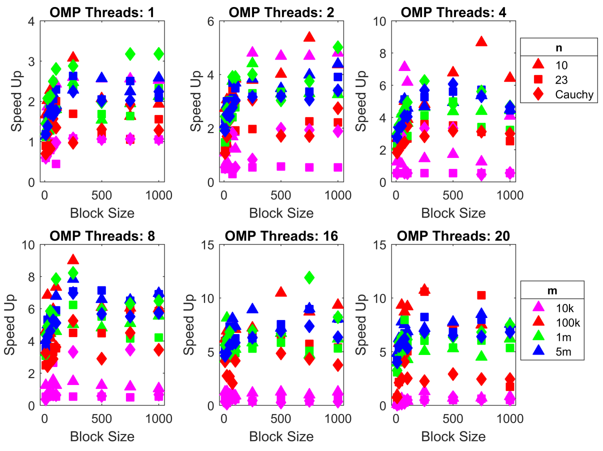
- Accept generic input,
- Validate the numerical result of the optimized routine,
- Measure performance of the original and optimized routines,
- Tune optimizations.

The generic input is read in from a file, where the file must contain at least one matrix A and resulting vector b. Should only one matrix and vector be supplied, the input will be duplicated for all instances of m. Validation of the optimized routine is performed by taking the difference of the output from the original and optimized routines. The mini-app will first compute the solution of the input using the original routine, and then the optimized routine. This way the output may be compared directly, and relative performance may also be measured using execution time. Should the optimized routine feature one or more parameters that may be varied, they are to be investigated such that the optimization may be tuned to the hardware. In this work, there is always at least one tunable parameter.

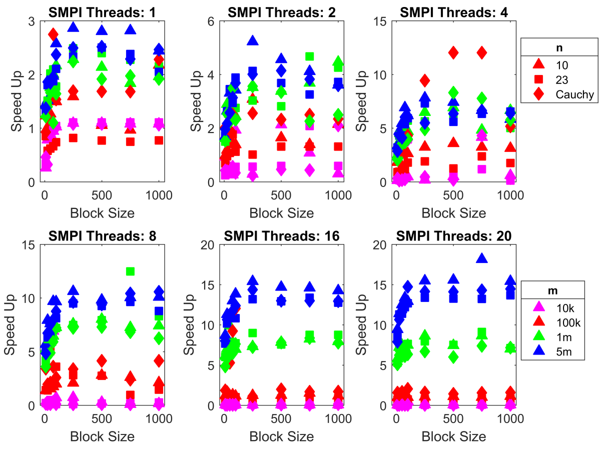
One feature that should have been factored into the mini-app design was modularizing the different versions of the Householder routine. In this work, two mini-apps were designed because each implements a different version of the parallel Householder routine; however, it would have been better to design a single mini-app that uses modules to include other versions of the parallel Householder kernel. With this functionality, it would be less cumbersome to work on each version of the kernel.

2.2 Parallel Householder

To parallelize the Householder routine, m is decomposed into separate, but equal chunks that are then solved by each *thread* – shared MPI tasks are equivalent to threads in this work for brevity. However, the original routine varies over m inside the inner-most computational loop (an optimization that benefits vectorization and caching), but the parallel loop must be the outer-most loop for best performance. Therefore, loop blocking has been invoked for the parallel sections of the code. Loop blocking is a technique commonly used to

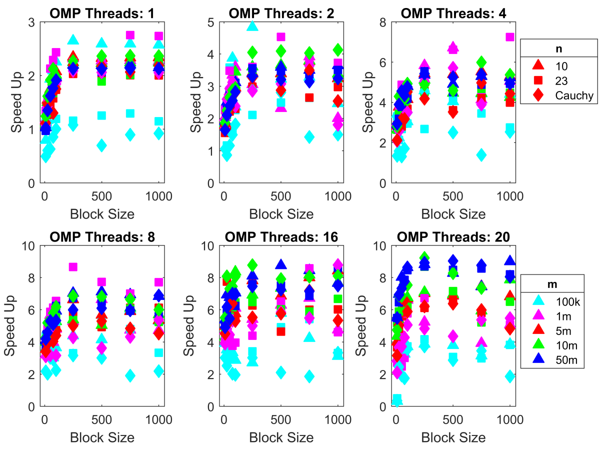


(a) MPI+OpenMP - 1 Node

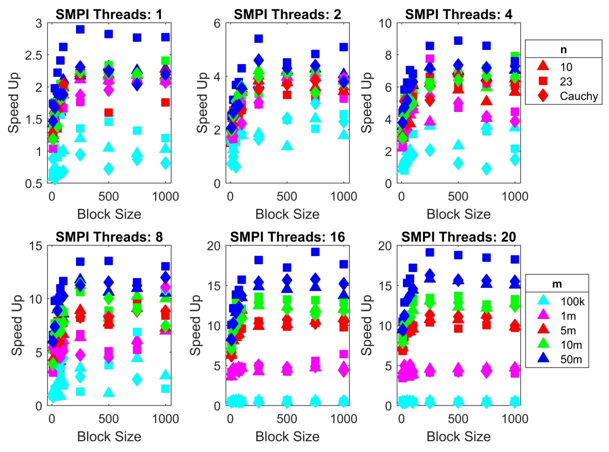


(b) MPI+SMPI - 1 Node

Figure 1: 1 Node Performance Evaluation: Speedup of the optimized mini-apps vs. the original routine with only 1 MPI task.



(a) MPI+OpenMP - 2 Nodes



(b) MPI+SMPI - 2 Nodes

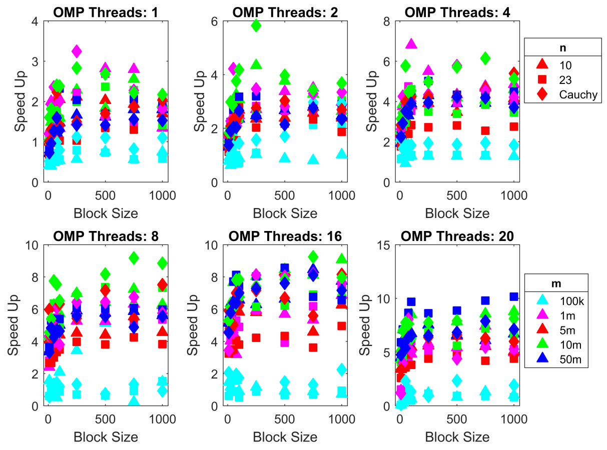
Figure 2: 2 Node Performance Evaluation: Speedup of the optimized mini-apps vs. the original routine with only 1 MPI task per node.

reduce the memory footprint of a computation such that it fits inside the cache for a given hardware. Therefore, the parallel Householder routine has at least one tunable parameter, block size.

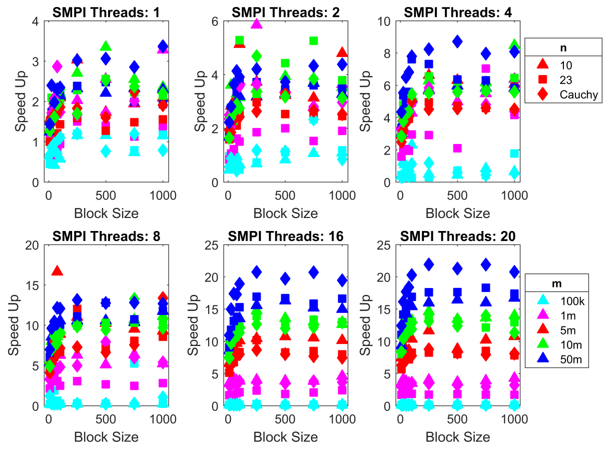
In this work, two flavors of the shared memory model are investigated: OpenMP and SMPI. The difference between OpenMP and SMPI lies in how memory is managed. OpenMP uses a public-memory model where all data is available to all threads by default. Public-memory makes it easy to add parallel statements, since the threads will all share this data, but threads are then susceptible to false-sharing, where variables that should otherwise be private are inadvertently shared. Shared MPI uses a private-memory model where data must be explicitly shared between threads, and all data is private by default. Private-memory makes any parallel implementation more complicated, because threads must be instructed to access specific memory for computation. Further, OpenMP creates and destroys threads over the course of execution which is handled internally and is costly to performance. SMPI threads are created upon execution start and persist throughout. This makes managing SMPI threads more difficult, since each parallel phase must be explicitly managed by the programmer. However, the extra work by the programmer may pay off in terms of performance, since less overhead is incurred by SMPI.

3 PERFORMANCE EVALUATION

This section presents the procedure and results of performance testing for the MPI+OpenMP and MPI+Shared MPI Householder Reflector kernel optimizations. For performance testing, it was of interest to vary the number of nodes used for the calculation because many nodes are often used when executing VULCAN with real-world simulations. Up to four nodes have been investigated in this work on a multinode HPC cluster. The number of MPI tasks and



(a) MPI+OpenMP - 4 Nodes



(b) MPI+SMPI - 4 Nodes

Figure 3: 4 Node Performance Evaluation: Speedup of the optimized mini-apps vs. the original routine with only 1 MPI task per node.

OpenMP threads are varied, as well as block size for loop-blocking in the parallel section.

For each mini-app, the optimized version of the Householder routine was validated against the original version by calculating the numerical difference in output. The validation found OpenMP to provide exact numerical solutions (a difference of zero) and SMPI had small numerical discrepancies (10^{-9}).

Computing Platforms. The performance evaluation has been conducted on a multinode HPC system *Turing* located at Old Dominion University [10]. Each node on Turing has dual-socket E5-2670 v2 (Ivy-Bridge) CPU's, each socket has 10 cores @ 2.5 GHz and 25 MB cache. A total of 64 GB RAM memory is available on each node. Up to four nodes are used and the network interconnect is Infiniband FDR (Fourteen Data Rate).

Results and Evaluation. The performance evaluation varies the size n for the input matrix and the number m of linear systems investigated. Two values of n, 10 and 23, are investigated, which are common sizes based on the VULCAN sample inputs. A third input, nicknamed `Cauchy`, for n is investigated, which is a square Cauchy matrix [4] of size 10. The number of linear systems m depends on the number of nodes. For the single-node performance tests, m is set to 10k, 100k, 1m, and 5m. For the multinode performance tests, m is set to 100k, 1m, 5m, 10m, 50m, and 100m. The number of threads was varied using powers of two: 1, 2, 4, 8, 16, and 20, because each node on Turing has a total number of 20 cores. The block size is varied using the values 10, 25, 50, 75, 100, 250, 500, 750, and 1000, in order to observe effects on the cache performance and memory latency.

The speedup for the single-node performance tests are shown in Fig. 1. Fig. 1a presents the MPI+OpenMP speedup and Fig. 1b presents the MPI+SMPI speedup where m, n, and `Block Size` are varied. Speedup is shown on the y-axis, block size on the x-axis, n is represented using a triangle for 10, square for 23, and diamond for Cauchy, and m is represented using colors as shown in the plot legend. Speedup for the multinode performance tests is shown in: Fig. 2 (2-node) and Fig. 3 (4-node). Notice that the workloads (m) are different for the multinode tests than for the single-node tests; 100k-50m vs. 10k-5m respectively.

Speedup is a measure of execution performance. A value of one means both versions have equal performance. A value less than one means the optimized version is worse than the original routine, and a value greater than one means the optimized version is better.

The workload, m=10k, is only investigated for the single-node case. Notice in Fig. 1 that speedup is consistently one or less. Therefore, the parallel Householder routine must have a sufficient workload to attain any speedup.

In all performance tests conducted, Figs. 1 to 3, Shared MPI consistently attains the greatest speedup over the original Householder routine. Speedup is normalized against the one thread per node performance for each respective workload (n and m) and block size. The maximum speedup for OpenMP is 12×, 9×, and 10×, and Shared MPI is 18×, 19×, and 23× for 1, 2, and 4 nodes, respectively. From the performance results, it is apparent that SMPI benefits from less overhead as a result of increased cost to the programmer.

It is interesting to note that the best performing input n varies for SMPI as the number of nodes varies. For one-node and the maximum number of threads, n of 10 has the best speedup. For two nodes, n of 23 has the best speedup, and n of Cauchy has the best speedup for the four-node case. This was an unexpected result, and one that is not obtained when using OpenMP. Further investigation is needed to determine if this is coincidence or a meaningful result.

Blocking performance, without shared memory parallelism, is captured by the one thread tests in Figs. 1 to 3. A max speedup of 3× is consistently measured no matter the mini-app and number of nodes. This finding shows that optimizing the algorithm for cache performance is mildly beneficial and should be considered for performance-bounded computational kernels.

4 CONCLUSION

In this work, mini-apps were developed to optimize the Householder Reflector kernel within NASA real-world application, VULCAN.

Two programming paradigms for shared memory parallelism were investigated, OpenMP and Shared MPI, and performance testing was conducted on a multinode system Turing for up to four nodes. Speedup, the measure of performance, was found to be higher for the Shared MPI version of the Householder mini-app than that for the OpenMP version. Specifically, the speedup for SMPI was up to $2\times$ that of OpenMP. With the maximum number of threads, SMPI obtains 23x speedup with sufficiently large workloads (m=50m). OpenMP was only able to achieve a speedup of $11\times$, which is about half of the expected speedup based on the number of threads used.

ACKNOWLEDGMENT

This effort was supported by NIA subaward activity 2B87 funded through the NASA Langley Computational Digital Transformation initiative and, in part, by the Turing High Performance Computing cluster at Old Dominion University. The authors would like to thank Nathan Gonda and Michael Poteat of Old Dominion University for their contribution.

REFERENCES

[1] F. Cappello and D. Etiemble. 2000. MPI versus MPI+OpenMP on the IBM SP for the NAS Benchmarks. In *Supercomputing, ACM/IEEE 2000 Conference*. 12–12. https://doi.org/10.1109/SC.2000.10001

[2] Martin J. Chorley and David W. Walker. 2010. Performance analysis of a hybrid MPI/OpenMP application on multi-core clusters. *Journal of Computational Science* 1, 3 (2010), 168 – 174. https://doi.org/10.1016/j.jocs.2010.05.001

[3] N. Drosinos and N. Koziris. 2004. Performance comparison of pure MPI vs hybrid MPI-OpenMP parallelization models on SMP clusters. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.* 15–. https://doi.org/10.1109/IPDPS.2004.1302919

[4] Miroslav Fiedler. 2010. Notes on Hilbert and Cauchy matrices. *Linear Algebra Appl.* 432, 1 (2010), 351 – 356. https://doi.org/10.1016/j.laa.2009.08.014

[5] R. Gerstenberger, M. Besta, and T. Hoefler. 2013. Enabling Highly-scalable Remote Memory Access Programming with MPI-3 One Sided. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, Article 53, 12 pages. https://doi.org/10.1145/2503210.2503286

[6] William Gropp, Luke N. Olson, and Philipp Samfass. 2016. *Modeling MPI communication performance on SMP nodes: Is it time to retire the ping pong test*. Vol. 25-28-September-2016. Association for Computing Machinery, 41–50. https://doi.org/10.1145/2966884.2966919

[7] Per Brinch Hansen. 1992. Householder Reduction of Linear Equations. *ACM Comput. Surv.* 24, 2 (June 1992), 185–194. https://doi.org/10.1145/130844.130851

[8] T. Hoefler, J. Dinan, D. Buntinas, P. Balaji, B. Barrett, R. Brightwell, W. Gropp, V. Kale, and R. Thakur. 2013. MPI + MPI: A New Hybrid Approach to Parallel Programming with MPI Plus Shared Memory. *Computing* 95, 12 (Dec. 2013), 1121–1136. https://doi.org/10.1007/s00607-013-0324-2

[9] T. Hoefler, J. Dinan, R. Thakur, B. Barrett, P. Balaji, W. Gropp, and K. Underwood. 2015. Remote Memory Access Programming in MPI-3. *ACM Trans. Parallel Comput.* 2, 2, Article 9 (June 2015), 26 pages. https://doi.org/10.1145/2780584

[10] HPC Group. 2016. Turing Community Cluster General Information. (2016). https://www.odu.edu/facultystaff/research/resources/computing/high-performance-computing.

[11] Message Passing Interface Forum. 2012. MPI: A Message-Passing Interface Standard Version 3.0. (2012). http://mpi-forum.org/docs/mpi-3.0/mpi30-report.pdf.

[12] Mikhail B. (Intel). 2015. An Introduction to MPI-3 Shared Memory Programming. (2015). https://software.intel.com/en-us/articles/an-introduction-to-mpi-3-shared-memory-programming.

[13] NASA. 2016. VULCAN-CFD. (2016). https://vulcan-cfd.larc.nasa.gov/.

[14] OpenMP. 2016. OpenMP: The OpenMP API specification for parallel programming. (2016). http://www.openmp.org/.

[15] R. Rabenseifner, G. Hager, and G. Jost. 2009. Hybrid MPI/OpenMP Parallel Programming on Clusters of Multi-Core SMP Nodes. In *2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*. 427–436. https://doi.org/10.1109/PDP.2009.43