A Path from Serial Execution to Hybrid Parallelization for Learning HPC

Tommy Franczak*†, Asare Nkansah‡†, Thomas Marrinan§† and Michael E. Papka†*
*Northern Illinois University, DeKalb, Illinois 60115
Email: {tfranczak,papka}@niu.edu
†Argonne National Laboratory, Lemont, Illinois 60439
Email: {tfranczak,nkansaha,tmarrinan,papka}@anl.gov
‡University of Kentucky, Lexington, Kentucky 40506
Email: asare_nkansah@uky.edu
§University of St. Thomas, St. Paul, Minnesota 55105
Email: tmarrinan@stthomas.edu

***Abstract*—Parallel and distributed computing are becoming necessary in almost all aspects of computation. Due to this growing demand, curriculum initiatives have been developed for integrating parallel and distributed computing into traditional undergraduate computer science programs. However, adoption has been slow resulting in many students lacking proper training for parallel and distributed computing. Two potential barriers for slow adoption are a deficiency in example programs that step students through the processes of parallelizing serial code, and the inaccessibility of dedicated machines to run highly parallel programs at scale within the confines of a course schedule. We have developed course material using a simple two-dimensional Lattice-Boltzmann Method Computational Fluid Dynamic simulation to walk students though shared memory parallelism, distributed memory parallelism, and hybrid parallel execution. We also created a custom mini-cluster comprised of 16 credit-card sized compute nodes, with a total of 288 cores, as an inexpensive solution for testing the scalability of different parallel models that can be deployed in a classroom setting.**

I. Introduction

Our society is quickly becoming more dependent on High-Performance Computing (HPC), with fields such as biology [1], finance [2], and renewable energy [3] utilizing computationally expensive simulations. These HPC simulations depend on running code that is massively parallelized and run on distributed compute systems. While many educational institutions have acknowledged the growing demand for HPC, there has been a lag on implementing curriculum changes to introduce students to parallel and distributing computing (PDC) methods.

The education of PDC should be a crucial component for undergraduate computer science programs. However, integrating these topics into an undergraduate curriculum is complicated to achieve effectively. There are not an abundant number of resources for HPC development that are compact enough to fit into the confines of a single course; even less that are self-contained applications with true authentic uses [4]. To complicate the matter, institutions encounter issues when attempting to educate students due to the availability and expense of cluster computing resources. The burden of scheduling a time shared resource and purchasing expensive hardware are often prohibitive for use in an undergraduate classroom [5].

Most computer science students are taught sequential semantics throughout their beginner years of programming. Therefore, we have developed a self-contained fluid dynamic application that can be run with serial execution, using shared memory parallelism, using distributed memory parallelism, or using a hybrid approach of both distributed and shared memory parallelism. For each execution method, students can be walked though the parallelization process and see correlations between the serial and parallel versions of the code, thereby making connections between a familiar paradigm and new concepts.

Even with quality material for teaching PDC concepts, many instructors still choose to only go as far as teaching shared memory parallelization due to the issues of scheduling and expense. While this is certainly better than completely ignoring PDC education, it still lacks teaching critical skills necessary in research and industry. In order to address the sometimes prohibitive nature of using a large shared resource in a classroom setting, we have built an inexpensive mini-cluster from credit card sized computers that can be administered by an instructor and dedicated to student use.

Our contributions to the field of PDC education that are highlighted in this paper are as follows:

- A proposed workflow for parallelizing simulations
- A proof-of-concept for an inexpensive mini-cluster comprised of credit card sized computers
- A self-contained software package, encasing a progression from serial execution to hybrid parallelism

II. Related Work

There are several publications that discuss methods for educating students on the various components of parallel programming. Rivoire [6] introduced a breadth-first parallel programming course targeted at shared memory machines. The goal of the course was to provide students with experience using a broad range of state-of-the-art programming models. Such courses can be a good introduction to parallel

computing and prepare student's to work on computationally intensive projects such as the one by Gowanlock et al. [7] who use conventional multi-core processors to improve the performance of their two-dimensional program. However, only covering shared memory parallelism can leave a significant void in PDC skills learned by a student.

In order to incorporate distributed memory parallelism into eduction, Eijkhout [8] addressed some of the fundamental issues with teaching MPI in a traditional manner. Spawning and joining threads within a shared memory application is very different conceptually from multiple processes all running simultaneously. Eijkhout notes three techniques that can be used to ease the transition for novice parallel programmers: introducing process symmetry using mpiexec on non-MPI code, demonstrating functional parallelism by programming unique behavior for different processes, and teaching collective operations. While these methods are a logical approach for teaching distributed memory parallelism, the authors still advocate for introducing these concepts with MPI, which can confuse students with unnecessary complexity for educational purposes.

Gardner and Carter [9] have developed Pilot as a library that abstracts MPI specifically for educational purposes. They point out certain confusing mechanisms inherent in MPI, such as ambiguous error messages. They also consolidate MPI's vast functionality down into a few dozen function calls. Their work does a good job of introducing an educational tool for distributed memory computing. However, when students are ready to engage with production code, there lacks a guide on how to transition from Pilot to MPI in order to leverage more advanced features.

Beyond the issues of teaching PDC concepts, Liu [5] explains the economic issues that come with parallel computing. In order to solve the problem, the authors proposed using Linux Beowulf clusters - commodity grade computers, linked together on a LAN. Although Beowulf clusters can be powerful, in most cases they require a large physical space to house the hardware. Depending on the quality of the machines, these can also be expensive. Harrell et al. [10] agree and establish the need for inexpensive HPC learning tools, while also calling for student made clusters. The mini-cluster proposed in this paper is our solution to the problems outlined by these authors.

III. Teaching PDC Concepts

Authentic use cases are extremely powerful tools when teaching any computer science concept, and PDC is no different. Unfortunately, there is currently a lack of such use cases, which is recognized by the HPC education community [11]. We have developed a Lattice-Boltzmann Method Computational Fluid Dynamic (LBMCFD) simulation as a coding example to help fill this void. The LBMCFD simulation was adapted from an online physics simulator by Daniel Schroeder [12]. This two-dimensional simulation provides students with a compact, self-contained program with sensible visual output. The LBMCFD simulation simulates two-dimensional fluid flow through an open-ended tube, where the fluid is impeded by barriers that can cause turbulence. Figure 1 provides a snapshot of what a single time step of the LBMCFD simulation could look like.

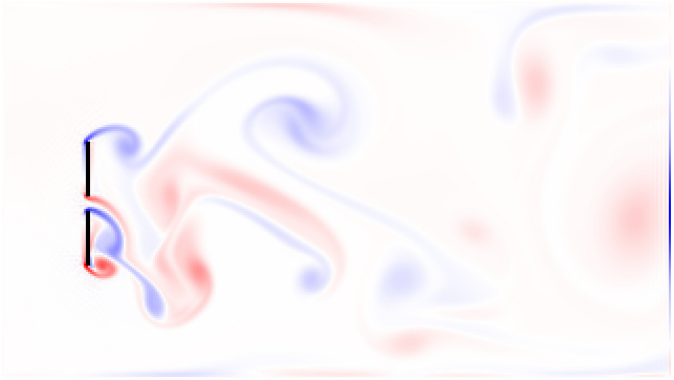

Fig. 1. Visualization of fluid vorticity produced from a single time step of the 2D Lattice-Boltzmann Method Computational Fluid Dynamic simulation (blue = spinning clockwise, red = spinning counter-clockwise).

In order to perform computations, the simulated physical space of the open-ended tube is discretized into a regular 2D grid, with each cell storing properties such as density, velocity, and vorticity. The 2D grid can easily be broken up into sub-grids with each computed upon nearly independently. Only peripheral information must be passed between neighboring sub-grids to update information at their shared boundaries. The same series of calculations is done repeatedly (each representing one time step in the simulation). Parallelizing the time steps is not possible due to the order dependent nature of these steps.

The following parallel implementations of the LBMCDF simulation can either be utilized as authentic sample code to showcase various PDC concepts to students, or they can be used as solutions for educators to use as the basis of programming assignments.

A. Shared Memory Parallelism

Shared memory is memory that may be simultaneously accessed by multiple processor threads within a given machine. Parallel threads share a global address space and can divide and conquer lengthy tasks. The advantage of shared memory is that no communication is needed between the processing elements, other than writing data to and reading data from the shared memory. This allows for time to be dedicated to computing.

One of the most popular interfaces that supports multi-platform shared memory multi-processing is OpenMP [13]. Using this thread-based interface's simple semantics, a programmer can enable multi-threading within their programs. The OpenMP library utilizes runtime library routines, compiler directives, and environmental variables in C, C++, and Fortran. The OpenMP compiler directives appear as comments within your code, and typically need to be enabled by a compiler flag. These compiler directives include functionality such as spawning and synchronizing threads.

Parallelizing programs with OpenMP can be thought of as function-level parallelization, where parallel blocks of code are located within the designated parallel directives. OpenMP programming has the inherent notion of a master thread. This is the processor thread that controls all of the non-parallel blocks of code, and is responsible for spawning the other threads.

We have developed an OpenMP implementation of the LBMCFD application. Each thread is given a partition of the simulation's 2D grid (a subset of rows that span the entire width). All threads simultaneously compute one time step for their sub-grid. Then the program synchronizes between time steps in order for the data to remain consistent.

B. Distributed Memory Parallelism

Distributed memory parallelism refers to running a multi-process application in which each process has its own private memory. By enabling communication between the processes, an overall application can solve a task in parallel. Distributed memory parallelism is important since it can theoretically scale indefinitely by using more computers, whereas shared memory parallelism is confined to the number of CPUs / cores within a given machine. The most common method for this communication between the different processors is using Message Passing Interface (MPI) [8]. MPI provides a mechanism for a program to communicate and exchange data between different processes in order to synchronize the actions of the program over its entire parallel execution. This method of communication is extremely practical and beneficial for a few reasons: the implementation is easily portable for different platforms that support the MPI standard, and it helps to maximize computational performance when implemented correctly. While MPI can be very useful, it can be difficult for parallel computing novices to comprehend.

Prior to using Eijkhout's model for teaching distributed computing with MPI [8] described in the related works section, we propose introducing students to distributed computing concepts using Pilot [9]. According to our workflow, students will have already been introduced to OpenMP at this point, and still have the idea of a master thread running the processes. MPI fundamentally abandons that idea and forces students to re-conceptualize parallelism, while also learning complex communication tools. Introducing the Pilot library instead will allow students to grow accustomed to the functionality of the MPI library, while still conceptually using a master process as well as reducing many other complexities.

1) Pilot Library: Pilot is an entry level message passing software that can be used as a pedagogical tool to introduce parallel applications [14]. Pilot was created to help programmers that are new to parallel programming avoid common parallel programming missteps. The structure of the parallel program is simplified, as the library focuses on function-level parallelism, similar to the shared-memory OpenMP library. Pilot also shares the OpenMP notion of a master thread, where the main process controls the actions of other separate processes.

In Pilot, the first step is to define the channels over which the programmer desires to perform communication. These communication channels are unidirectional, forcing communications to flow only one way. PI_Read() and PI_Write() do the bulk of communications, with the first argument being the specified channel. What makes the read and write so valuable for students is the usage of string formatting to specify data types and quantities, similar to a C printf() statement. For example, the read and write functions are as follows:

```
PI_Read([channel variable], [format string],
   [list of variable addresses])

PI_Write([channel variable], [format string],
   [list of variable values])
```

Pilot keeps its library simple by using three arguments per function, with the second argument specifying the type of the data being communicated using string formatting. In PI_Write, the following arguments are simply the values of the variables needing to be communicated. In PI_Read, the following arguments are the addresses of the variables being received. What is unique to the Pilot interface are the universal parameters for all communication functions. Understanding a single function allows the student to replicate the format with the rest of the library.

The Pilot library can help students learn in other ways as well. It includes verbose error messages that are simple to understand. Rather than decrypting MPI's vague errors messages, students are presented with comprehensible statements, such as "Communication channel already exists." This library also excludes a barrier; a tool that allows explicit synchronization of different ranks. The HPC consortium considers these to be a crutch for MPI newcomers [9], and eliminating them results in students solving these communication issues with logic instead.

We have implemented a Pilot version of the LBMCFD application. Each process is given a partition of the simulation's 2D grid in a similar manner as the OpenMP version (a subset of rows that span the entire width). All processes simultaneously compute one time step for their sub-grid. Then the program synchronizes between time steps by performing inter-process communication in order for the data to remain consistent.

2) MPI: MPI is a standardized message passing interface which deals with multiple instruction, multiple data (MIMD) programming [15]. By this point, students have grasped the basic concepts of distributed memory parallelism, but they have not worked with MPI's more advanced functions. It must be stressed during this point that after initializing MPI, every process acts in parallel, and the concept of a master thread is gone. Other structures within an application are now simply replaced by their MPI counterparts, such as MPI_Send() and MPI_Recv() replacing PI_Write() and PI_Read().

In order to teach the impact that inter-process communication can have on overall performance, we compare different communication patterns within the LBMCFD application. It is important for students to understand that there is more than one way to transfer data between processes and that certain techniques will perform more optimally. This can be a valuable exercise for students to either implement or run tests using the final code in order to measure the difference in execution time based on communication pattern.

Splitting up the 2D grid for a distributed memory application results in artificial "boundaries" being created in the overall data domain. Therefore each of processes must exchange some data with its immediate neighbors in order to properly complete its portion of the simulation. In an MPI application with N processes, each process is assigned a rank (unique value from 0 to (N-1)). The rank number is used to determine which process is computing over each sub-grid in the overall domain and allows each process to determine which other processes it needs to communicate with.

Since the LBMCFD simulation deals with a 2D grid, we have provided the following patterns for breaking up the domain for MPI execution:

1) Horizontal Rows
2) Vertical Columns
3) Rectangular Grid

Figure 2 provides an example of how the LBMCFD 2D grid data can be distributed using these three patterns.

Two-dimensional arrays are used very regularly in complex MPI-based programs. These arrays can be used to extract specific information within a program and transfer that same information across processors to keep the simulation updated and synchronized as it progresses. One of the most common methods for communicating the data is through the MPI send-receive call:

```
MPI_Sendrecv(const void *sendbuf, int
   sendcount, MPI_Datatype sendtype, int dest,
   int sendtag, void *recvbuf, int recvcount,
   MPI_Datatype recvtype, int source, int
   recvtag, MPI_Comm comm, MPI_Status *status)
```

This call is extremely similar to the separate send and receive calls discussed in the previous section of this paper. The only difference is that the calls are combined together to make one. Using this call, we can discuss how the information within the horizontal rows are communicated.

3) Horizontal Rows: There are many factors that can effect the performance of a distributed memory application. One important factor is the manner in which data is split amongst the processes. One of the simplest methods of distributing data for the LBMCFD simulation is by splitting it horizontally into consecutive rows so that each process uses contiguous chunks of data.

While this technique for exchanging data will work correctly, it also has its flaws. Since boundaries between each rank consists of an entire row in the overall 2D grid, the total amount of data transferred per time step would grow as the application uses more processes. This can cause significant communication overhead that is independent of scaling up the grid size.

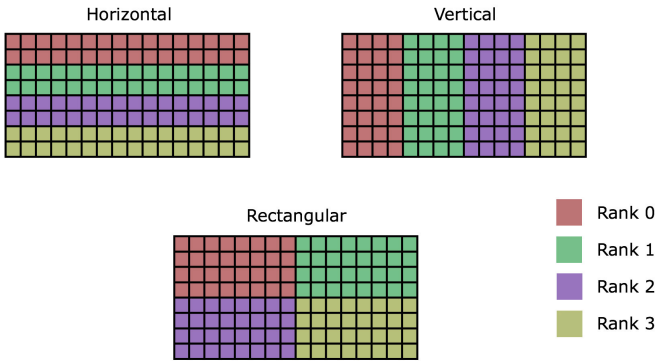

Fig. 2. Data layout illustration for distributing data into horizontal rows, vertical columns, and a rectangular grid when using 4 processes.

Even though this method may not be the most efficient, we feel that it is important for students that are working with complex simulations to start with this pattern. It is relatively easy to grasp and implement since the data being transferred is stored in contiguous memory. Additionally, MPI code using horizontal rows can serve as a template for students to implement other communication patterns.

4) Vertical Columns: The LBMCFD data can also be distributed in vertical columns. This is conceptually similar to horizontal rows, but will expose students to dealing with data that is not stored contiguously in memory. The following MPI call be used to specify a subarray that will be used for exchanging boundary information (in this instance, a single column of data):

```
MPI_Type_create_subarray(int ndims,
   const int array_of_sizes[], const
   int array_of_subsizes[], const int
   array_of_starts[], int order, MPI_Datatype
   oldtype, MPI_Datatype *newtype)
```

After specifying the data that the programmer would like to transfer, we can use the send-receive communication call to exchange the columns of data between neighboring ranks.

Unfortunately, distributing data in vertical columns does not inherently improve communication efficiency. The same issue of increased total amount of data transferred as more processes are used will persist with the vertical layout. However, if the overall 2D domain is wider that it is tall, the application should experience a reduction in communication overhead since the size of a single column is less than the size of a single row.

We believe that this communication pattern is also crucial to the process of understanding complex MPI programs. Both trial-and-error and incremental improvement can serve a crucial component to understanding how MPI-based programs behave and what techniques can be applied to optimize the

code. Also, both the horizontal rows and vertical columns serve as important stepping stones for configuring the more optimized rectangular grid distribution correctly.

5) Rectangular Grid: The rectangular grid distribution results in the the most complex communication patterns due to the fact that there are more boundaries to deal with between the ranks. In both the horizontal and vertical patterns, each rank could have at most two boundaries on either side of them. With the rectangular grid, boundaries are being exchanged horizontally, vertically and diagonally (maximum of eight neighbors). In order to properly exchange the boundaries, a combination of the MPI functions that were described for the prior two patterns will have to be utilized. The MPI_Type_create_subarray() call will have to be used to specify the data to transmit between neighbors on the right or left, whereas basic send and receive calls using contiguous memory can used to transmit between neighbors above, below, and diagonally.

Although crafting the rectangular grid is more complex, it results in the most efficient method of communication between processes. While there are more communication calls to be made via MPI with the rectangular grid, there will be significantly less data to transfer in total. This results in more efficient computing performance. This method sees optimal communication benchmarks when the number of ranks is a perfect square (e.x. 4,9,16). This causes there to be less communication boundaries entirely.

In both the vertical and horizontal simulations, the amount of data transferred is equal to one less than the amount of ranks, multiplied by length of the respective width/height. Therefore, as we increment the number of ranks by one, we see the transfer size incremented linearly by one width/height of the simulation. However, in the Rectangular Simulation, we do not have a linear increase in total transfer size. To calculate the total amount transferred, we must figure out how many vertical and horizontal slices there are, and multiply those values by their respective dimension. To do this, we must calculate the two factors closest to the square root of the number of ranks - then, we subtract one from each factor. These new factors are the number of both vertical and horizontal lines. After multiplying the amount of horizontal and vertical lines by their corresponding dimensions, we also need to account for the diagonal communications; however, these are only one element in size. This number is obtained by multiplying the amount of vertical lines by the amount of horizontal lines, and then multiplying by two to account for the bi-directional diagonal communications.

C. Teaching Hybrid Parallelism

Hybrid Parallelism is a concept that involves integrating both shared and distributed memory parallelism. This is an important concept to take advantage of many different factors. When dealing with programs of massive scale, distributed memory communications can cause bandwidth issues [16]; so cutting down on communications is imperative. MPI + X models are crucial for coordinating the communication and the computational parts of an application [17]. Localizing parallelism into different threads, rather than MPI processes, will significantly cut down these communications [16]. Shared memory interfaces obviously fail when cross-node communications are required, so they are unable to take advantage of computing clusters. Also, the shared memory parallelism is limited by the amount of threads that can be embedded into a single machine with a shared memory. A combination of the two will allow for the best of both worlds: Cross-node processes with limited communications.

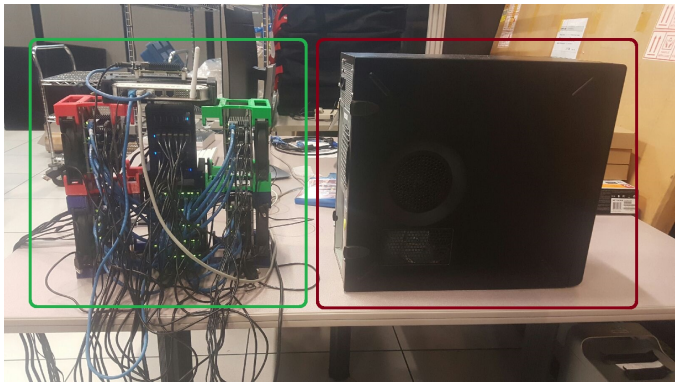

Fig. 3. The mini-cluster (highlighted inside the green box) juxtaposed next to a traditional desktop computer (highlighted inside the red box)

MPI + X is a popular model of Hybrid Parallelism. The X variable represents a secondary interface used for both cutting down total communications, and for taking advantage of certain hardware. In William Gropp's, "Is MPI+X Enough for Exascale?" powerpoint, he states that, "MPI is a vital component in the Exascale stack," while also noting that more effort needs to be placed on the X [18]. Since we required threading in our case, we implemented MPI + OpenMP. Seeing as OpenMP is our 'X', his research supports our workflow, as learning OpenMP first will create an emphasis on it. MPI still handles the communication methods associated with distributed memory, and OpenMP spawns threads that handle the shared memory calculations.

In regards to our specific implementation, we start by having each rank calculate the size of its partition of the 2D LBMCFD array. The rank will then allocate its own partition into its own memory. The rank will then be responsible for only that partition of the object, and as such only does calculations within the bounds of it. Furthermore, OpenMP will then spawn the threads and assign a collection of rows to each individual thread. In total, the amount of processing elements being used will be equal to the amount of nodes multiplied by the amount of threads spawned.

IV. Pedagogical Tools

Undergraduate education of HPC faces many complications, including the cost and space of an HPC system [10]. We implemented a mini-cluster comprised of 17-Parallela microcomputers to develop and execute our code. This mini-cluster, as seen in Figure 3, contains one head node, and 16

slave nodes. The choice of Parallella boards was due to their capabilities of both cross-node and on board parallelism. This microcomputer was selected because it has a 2-core Intel ARM processor and a 16-core Epiphany co-processor [19].

The ARM processor is capable of on-board parallelism by offloading programs onto the 16-cores of the Epiphany co-processor. The ARM processor supports both OpenMP and MPI parallelization. The Epiphany co-processor currently supports OpenMP and has a custom implementation of MPI under development [20]. For our examples, however, we decided against using the co-processor. This was for a variety of reasons, but mainly due to the lack of on core memory. Each core of the Epiphany co-processor is limited to 32KB of memory, which would only support a low resolution version of the LBMCFD simulation.

This mini-cluster takes up less space than a typical desktop computer, and could assimilate into a classroom setting. The boards are also inexpensive, with a sticker price of $99 [21]. However, since our work did not leverage the advanced co-processor technology, significantly cheaper microcomputers could be used to achieve the same purpose [22] [23]. It would be our recommendation to use alternative microcomputers to build a mini-cluster for teaching undergraduate PDC concepts. However, a mini-cluster made with Parallela boards could still be useful in education for an advanced PDC course that would leverage the co-processors.

While the microcomputers cannot achieve the same performance as traditional desktops, they do offer great scalability. This is a useful tool for highlighting the benefits of parallelism in a classroom setting. Additionally, the mini-cluster can serve as a development platform for true HPC applications. We successfully developed the LBMCSF simulation on the mini-cluster, then transferred the code to Argonne National Laboratory's Cooley cluster, which is comprised of 126 nodes each with 12 cores and 384GB of memory.

V. Results

After evaluating the results of our proposed methodology, our research proved both the overall performance impact of using parallel and distributed computing while presenting an effective alternative to educate people on the matter.

The shared memory oriented program that we discussed previously in the paper was tested with steadily increasing amount of threads. The test proved our hypothesis that an increase in the thread count would lead to a direct, positive correlation to the overall performance of the simulation. Results of this test would of course depend on how many threads a given computer has on standby. In this performance test, we showed that a computer with 12 threads would be at its peak potential when all of them were utilized in sync.

For the distributed memory parallelism, we started by testing the performance of the different MPI communication patterns as proposed earlier in the publication. As shown in Figure 4, in comparison to the vertical and horizontal communication types, the simulation using the rectangular grid has a remarkably low communication time. These results

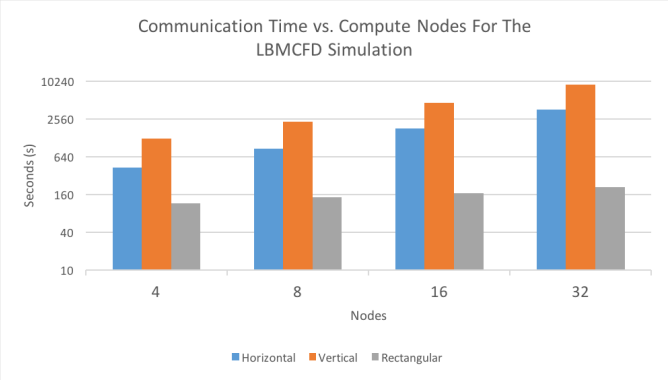

Fig. 4. Average communication time based on data distribution pattern for a 4525 x 2263 LMBCFD simulation with 60,000 time-steps using a varying number of MPI Ranks.

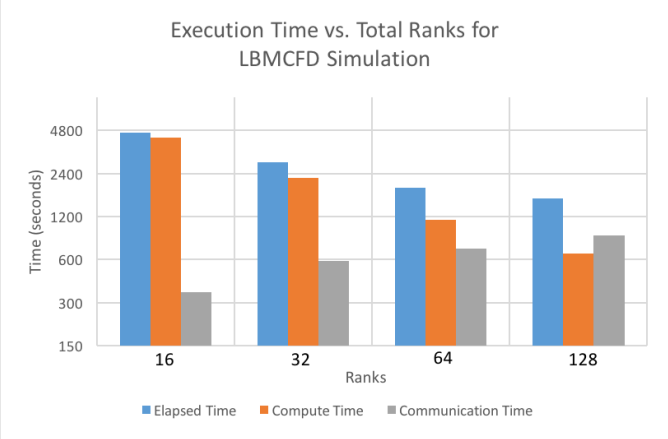

Fig. 5. Average execution time of a 4525 x 2263 LMBCFD simulation with 60,000 time-steps using a varying number of MPI ranks.

further proved the benefits of using the slightly more complex communication pattern since the amount of total data being sent is significantly less. After discovering the most efficient communication pattern, we tested the rectangular grid MPI program over a steadily increasing amount of ranks, as shown in Figure 5. According to our results, the communication, compute, and overall elapsed times all steadily decreased until about 128 ranks. We also determined that although increasing the amount of ranks usually results in a decreased elapsed time, once the amount of ranks reaches a certain amount the communication time is far too long to continue to be effective for the overall elapsed time – thus slowing down the program's efficiency.

The most interesting part of the results in our opinion are the overall elapsed times for the hybrid paralleled simulation shown in Figure 6. We observed that if we increased the number of nodes that were running the program the timings substantially decreased. This decrease was far superior to both the distributed memory and shared memory parallelism when they work alone; proving that if they work in harmony, the performance will be at is optimum efficiency.

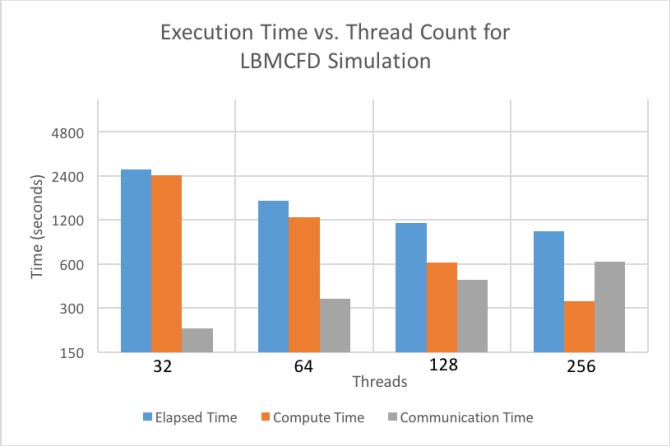

Fig. 6. Average execution time of a 4525 x 2263 LMBCFD simulation with 60,000 time-steps using a varying number of MPI Ranks and utilizing eight threads per rank.

VI. Conclusion

This paper proposes an educational model that follows an intuitive progression from serial execution to hybrid parallelism. We discuss the importance for HPC at an undergraduate level, and provide a self-contained software package, a proposed work flow, and an inexpensive alternative to traditional HPC systems. Since the Parallella board is designed for more advanced PDC concepts, a similar mini-cluster built from other microcomputers, such as the Raspberry Pi or ODROID-XU4, may be better suited for undergraduate education.

Results from the LBMCDF application highlight how hybrid parallelization provided the best performance when running on production supercomputing systems. As computing moves towards exascale, every performance optimization matters. Teaching students PDC concepts using an incremental workflow all the way through hybrid parallelism is imperative for producing an HPC workforce capable of pushing applications to their peak performance.

Acknowledgment

This research used resources of the Argonne Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC02-06CH11357. The simulation source code is available with permission from Argonne National Lab at https://www.xgitlab.cels.anl.gov/fl/lbmcfd.

References

[1] B. Langmead, "Practical software for big genomics data," in *2013 IEEE 3rd International Conference on Computational Advances in Bio and medical Sciences (ICCABS)*, June 2013, pp. 1–1.

[2] M. Smelyanskiy, J. Sewall, D. D. Kalamkar, N. Satish, P. Dubey, N. Astafiev, I. Burylov, A. Nikolaev, S. Maidanov, S. Li, S. Kulkarni, C. H. Finan, and E. Gonina, "Analysis and optimization of financial analytics benchmark on modern multi- and many-core ia-based architectures," in *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, Nov 2012, pp. 1154–1162.

[3] A. Papavasiliou, S. S. Oren, and B. Rountree, "Applying high performance computing to transmission-constrained stochastic unit commitment for renewable energy integration," *IEEE Transactions on Power Systems*, vol. 30, no. 3, pp. 1109–1120, May 2015.

[4] N. Giacaman, S. Kalra, and O. Sinnen, "The active classroom: Students and instructors parallel programming in parallel," in *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*, May 2015, pp. 739–745.

[5] J. Liu, "20 years of teaching parallel processing to computer science seniors," in *2016 Workshop on Education for High-Performance Computing (EduHPC)*, Nov 2016, pp. 7–13.

[6] S. Rivoire, "A breadth-first course in multicore and manycore programming," in *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, ser. SIGCSE '10. New York, NY, USA: ACM, 2010, pp. 214–218. [Online]. Available: http://doi.acm.org/10.1145/1734263.1734339

[7] M. Gowanlock, D. M. Blair, and V. Pankratius, "Optimizing parallel clustering throughput in shared memory," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 9, pp. 2595–2607, Sept 2017.

[8] V. Eijkhout, "Teaching mpi from mental models," in *2016 Workshop on Education for High-Performance Computing (EduHPC)*, Nov 2016, pp. 14–18.

[9] W. Gardner and J. Carter, "Using the pilot library to teach message-passing programming," in *2016 Workshop on Education for High-Performance Computing (EduHPC)*, Nov 2016, pp. 6–10.

[10] S. L. Harrell, H. A. Nam, V. G. V. Larrea, K. Keville, and D. Kamalic, "Student cluster competition," *Proceedings of the Workshop on Education for High-Performance Computing - EduHPC 15*, 2015.

[11] D. Bunde, "Peachy parallel assignments session of eduhpc-17," private communication, 2017.

[12] D. Shroeder, "Fluid dynamics simulation." [Online]. Available: http://physics.weber.edu/schroeder/fluids/

[13] "The openmp api specification for parallel programming." [Online]. Available: http://www.openmp.org/

[14] "Easy as pilot." [Online]. Available: http://carmel.socs.uoguelph.ca/pilot/

[15] C. The MPI Forum, "Mpi: A message passing interface," in *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '93. New York, NY, USA: ACM, 1993, pp. 878–883. [Online]. Available: http://www.ulib.niu.edu:4064/10.1145/169627.169855

[16] U. S. Wickramasinghe, G. Bronevetsky, A. Lumsdaine, and A. Friedley, "Hybrid mpi - a case study on the xeon phi platform," *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers - ROSS 14*, 2014.

[17] R. F. Barrett, D. T. Stark, C. T. Vaughan, R. E. Grant, S. L. Olivier, and K. T. Pedretti, "Toward an evolutionary task parallel integrated mpi + x programming model," in *Proceedings of the Sixth International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM '15. New York, NY, USA: ACM, 2015, pp. 30–39. [Online]. Available: http://www.ulib.niu.edu:4064/10.1145/2712386.2712388

[18] W. Gropp, "Is mpi+x enough for exascale?" pp. 1–28, 2014. [Online]. Available: https://www.pdc.kth.se/groppexampi14

[19] A. Olofsson, T. Nordstrom, and Z. Ul-Abdin, "Kickstarting high-performance energy-efficient manycore architectures with epiphany," *2014 48th Asilomar Conference on Signals, Systems and Computers*, 2014.

[20] "Programming epiphany/parallella with coprthr-2." [Online]. Available: http://www.browndeertechnology.com/coprthr2.htm

[21] "Parallella." [Online]. Available: https://www.parallella.org/

[22] K. Doucet and J. Zhang, "Learning cluster computing by creating a raspberry pi cluster," in *Proceedings of the SouthEast Conference*, ser. ACM SE '17. New York, NY, USA: ACM, 2017, pp. 191–194. [Online]. Available: http://doi.acm.org/10.1145/3077286.3077324

[23] "Low cost odroid xu4 compute cluster." [Online]. Available: http://diybigdata.net/odroid-xu4-cluster/