Performant, Portable, and Productive Parallel Programming With Standard Languages

Michael Wolfe, *NVIDIA Corp, Hillsboro, OR, 97006, USA*

The perfect solution to the P^3 (performance, portability, productivity) problem is a single version of an application that gives high performance across a wide range of target systems and is easy to develop and maintain. Actual solutions give up some level of performance, portability, or productivity, or all three. Here we review three periods in the past 65 years when the P^3 problem had good solutions. But it is harder today, with greater parallelism. We propose and argue in favor of a machine model to help programmers design algorithms and data structures that will exhibit performance portability. We then discuss the main point, in what language to express a parallel program to get all three of high performance, wide portability, with good productivity. We propose that the community, including both applications developers and language implementers, should focus on the parallel features in existing and future standard languages.

The Performance, Portability, Productivity (P^3) problem is generally a tradeoff. How much performance are we willing to give up to only maintain a single version of a program across multiple different target systems? Or, how much extra effort are we willing to expend in order to achieve higher performance on different computer architectures? Or, what machines will we focus on, and what target systems will we abandon, to simplify the effort required to tune performance?

Throughout the history of supercomputing, higher performance mostly came from clock rate and parallelism. Early supercomputers introduced pipeline parallelism, instruction-level parallelism, and out-of-order instruction execution, all using hardware to extract parallelism from a sequential instruction stream. A number of vector processors emerged, using special instructions that operated on streams of operands from memory or vector registers, implemented with multiple functional units in SIMD mode or pipelined functional units. Shared-memory multiprocessors followed as well. We see elements of all these types of parallelism in current multicore CPUs, with aggressive instruction dispatch functionality, pipelined functional units, SIMD operations, and now dozens of cores each of which can support multiple simultaneous threads in hardware.

Supercomputers in the 1990s scaled to higher processor counts by using many nodes, where each node initially had a single processor. Supercomputers today are designed such that each node has one or more multicore CPUs with all those levels of parallelism. A growing number include one or more GPU accelerators, introducing another type of parallelism to the mix. An important question facing the community today is how we should express a parallel program for the supercomputers we have today and those we see coming in the future.

At the lowest level, pipeline and instruction-level parallelism are mostly managed by the hardware, with compilers generating optimized code to exploit them. Programmers rarely have to change their applications to benefit from pipeline or instruction-level parallelism, except for some innermost kernels where some programmers might manually unroll the inner loops.

At the highest level, MPI is commonly used to implement parallelism across nodes, expressed as

Digital Object Identifier 10.1109/MCSE.2021.3097167
Date of publication 19 July 2021; date of current version 23 September 2021.

MPI library calls to get an MPI rank identifier in each process, and to communicate and synchronize between the processes. The rank identifier is used in the process to determine what partition of the data to allocate and process and what subset of the computation to perform. MPI itself does not help with that determination, though there are frameworks built on MPI that do. MPI is portable and performant because every vendor has an optimized MPI implementation for its particular processor and interconnect, and any frameworks built on MPI inherit that portability and performance. An advantage of MPI parallelism is that it looks the same across all target systems; there are few target-specific decisions to be made, and program portability is not an issue, though sometimes process placement is critical to the best performance. MPI has also been used to implement flat parallelism across nodes and across cores and threads within a node. A flat MPI-only parallelism model is more productive than an MPI+X two-level parallelism model, but it gives up potential performance in three ways. It is hard, if even possible, to take advantage of the shared memory between threads on a node when the two threads are controlled by MPI message passing. Also, MPI-only parallelism would not take advantage of SIMD-parallel operations, so there are always at least two levels of parallelism that need to be designed into the algorithm. Finally, MPI does not manage accelerators, so those need to be handled some other way.

Perhaps the most common X in MPI+X today is OpenMP, which in the latest version can control all the node-level parallelism: threads, SIMD, and accelerators. But like most current parallel programming models, it conflates expressing parallelism with exploiting parallelism. The commonly used OpenMP *omp parallel for* construct does not actually say that the loop iterations are data independent and can safely be executed in parallel with whatever parallelism mechanisms are available. It says to create a team of threads and to distribute the iterations across those threads in a defined manner, and that any data races or conflicts between those threads have been satisfied by the programmer. Therefore, a compiler cannot use an *omp parallel for* directive to determine that it is safe to vectorize the loop iterations using SIMD instructions. Instead, the programmer must add the *simd* directive. Now the programmer has to determine not only which loops are parallel, but also which loops to distribute across threads and which to execute in SIMD mode. It is enough of a challenge to determine that a loop is actually parallel, it is expecting a lot to require programmers to make code-generation decisions for every parallel loop in the application, and then to revisit those decisions for every target system. Other recent node-level parallelism frameworks (Kokkos,[1] RAJA[2]) suffer the same portability and productivity drawbacks.

Nevertheless, we argue that we can arrive at a good P^3 solution, across a wide range of computer architectures. We start by reviewing a few historical points in time where we had good P^3 solutions, and what was common among them.

P^3 IN EARLY SUPERCOMPUTING

In the earliest days of computing, software was written in machine language. The very first compiler was delivered by IBM for the Fortran language in 1958, over 60 years ago, for the IBM 704.[3] The language and its compiler were designed to approach the performance of hand-tuned machine code. The efforts at IBM to generate efficient code spawned the field of compiler optimization out of thin air, including developments such as control flow graphs, data flow analysis, induction variable recognition, register allocation, and more. It unknowingly set the bar for the P^3 problem. Fortran programs were decisively more productive than machine code, though that is a low threshold to beat. Fortran programs were also remarkably performant, largely because the IBM compiler team designed the language to allow for compiler analysis and designed the compiler to take advantage of that. And Fortran programs became portable, because IBM did not prevent other vendors from implementing Fortran, which became the first standardized programming language in 1966.

The reason that Fortran solved the P^3 problem reasonably well in those days was that the machine performance model was similar across different computer systems. Many of the architectural details that we worry about today (data cache locality, instruction cache reuse, SIMD operations) did not exist. A programmer interested in performance needed mostly to reduce the number of instructions and memory operations, and that would improve performance on any target system.

Another innovation in supercomputers came about 20 years later with the introduction of vector instructions. The Cray-1 introduced vector registers and vector instructions, similar in many respects to the SIMD registers and instructions available in many current microprocessors. Many people forget that the Cray-1 was initially successful because it was the fastest scalar computer available at the time, with twice the clock speed of the then-current champion (the Control Data 7600), but if your program could use the

vector operations, it could see a performance boost of up to five times. This was enough to convince many programmers to reorganize their data structures and algorithms to be amenable to vector processing. Unlike some contemporaneous vector machines, the Cray system depended heavily on an automatic vectorizing compiler to generate vector instructions.[4] The automatic vectorization capabilities in the Cray compiler were not particularly impressive compared to those being developed in academia at the time, but it was very effective largely because of its vectorization report. The report would tell the programmer which loops did and did not vectorize, and if not, why not, often with quite detailed and precise messages. This allowed the programmer to modify the code in the loop to avoid the hindrance to vectorization, or, in some cases, to override the compiler analysis and tell it to generate vector code anyway.

The impact of this was another solution to the P^3 problem for vector computers. The vectorizing compiler allowed programmers to achieve vector performance. It also allowed them to be more productive than having to write machine language subroutines or special syntax (such as vector syntax and *Q8* calls used in the Control Data STAR-100 and successors). Other vendors developed vector processors, including IBM, NEC, Fujitsu, Hitachi, Convex, even a vector Digital VAX, and each vendor also implemented a vectorizing compiler to at least match the capabilities of the Cray compiler. This allowed vector programs to achieve portability across a wide variety of vector computers of the time. Again, the reason that the vectorizing compiler solved that P^3 problem reasonably well was that the machine performance model was similar across many of these vector processors, and all the vendors developed appropriate compilers. Programming for vector operations on the inner loops and optimizing memory access for stride-1 was more than half the performance problem on any of these target systems. Vectorizing compilers allowed programmers to remain within the standard language features.

About 20 years later, the computer industry had progressed to the point where developing and producing a new computer processor was becoming too expensive for any low volume business. This meant that supercomputer-specific processors were unprofitable and essentially abandoned. It was much less costly to buy an array of commercial processor systems and combine them with a network. Many strategies for programming across the network were explored, and in the mid 1990s the message passing interface (MPI) was documented by the MPI Forum. MPI is implemented and presented as a library, and intended to be the low-level interface for communication between computer nodes across a network. There are a number of higher level frameworks built on MPI, but none have achieved the level of standardization and adoption of MPI itself. MPI was particularly successful because it hid many of the details of the network from the program, such as the protocol and topology.

The impact of MPI is another P^3 solution for the supercomputers available from the mid 1990s for the next 15 or 20 years. Some may argue that using MPI or any framework based on MPI is not intrinsically a highly productive programming environment, but it is certainly more productive than programming in any lower level framework. Portability is achieved because every supercomputer or network vendor has an optimized implementation of MPI taking advantage of the features of their computer or network interface. And performance is achieved when the programmer optimizes for minimal messages and synchronization, and for proper data and work distribution across the available nodes. An MPI program could perform well across a wide variety of message passing supercomputers.

There were language-level approaches proposed and implemented for managing network-level parallelism, such as High Performance Fortran, but they mostly failed. One reason is that making a bad synchronization or communication decision was very costly, and a compiler or runtime often made poor decisions. Another was that different system vendors had widely different efforts for such an approach, so even if one vendor had a good implementation, code written for that system was not portable to other systems. PGAS languages, including Fortran images and coarrays, address these issues by exposing data placement and communication to the programmer, basically at the same level as MPI.

SUCCESSFUL P^3

Each of the three P^3 points mentioned above were developed in response to a new computer architecture. They were successful because the various target systems had a similar performance profile, and the system vendors had the desire and will to provide performant implementations of the appropriate system software. Fortran and other programming languages were introduced at the dawn of digital computing, to hide the complexity of machine language and differences between machines. Vectorizing compilers were introduced to exploit vector parallelism in the hardware, and to hide the vector instruction details and

the differences between vector instructions on different machines. MPI was created to standardize control of parallelism across a network of homogenous nodes, to hide the network topology and communication transport details.

Programmers still had to change and tune their programs to expose and optimize the right kinds of parallelism. To get the advantage of vector computers, programmers had to retune their algorithms to expose vector parallelism and rewrite parts of their programs. They could either write SIMD intrinsics, or take advantage of a vectorizing compiler on the inner loops. In either case, the program needs the right data structure and the right algorithm to expose the right kind of operations that benefit from SIMD instructions. The question was how to express that SIMD parallelism. SIMD intrinsics require the programmer to think about SIMD register length and count, which can differ between vendors and even between generations of processors from the same vendor. Consider those programmers who recoded their Intel SSE intrinsics to AVX intrinsics when the Intel Sandy Bridge processors were introduced, only to have to recode them again using AVX-512 intrinsics for the Intel Knights Landing and Skylake processors. By depending on vectorization, the rewriting and tuning effort pays off across a wide range of target systems. By providing similar vectorization capabilities, the vendors and compilers played an important role in the P^3 solution.

Similarly, to take advantage of clusters, programmers had to rearchitect their programs at a high level to compute over a subset of data on each node, and to communicate among an array of cooperating nodes. Because the performance models for different clusters were similar, a rewrite to optimize or tune performance for machine A was most likely also a good decision for machines B and C.

A P^3 solution is successful if it achieves good performance, easy to use, and programmers do not have to write target-specific code. Here I propose a machine performance model that I argue characterizes the important performance elements of a wide range of current and future supercomputers. The model will be useful when designing P^3 algorithms and programs by focusing on the important features that affect performance regardless of the specifics.

P^3 Machine Model

The proposed machine performance model is based on four levels of parallelism: network, thread, SIMD, and accelerator, and the associated data management. For network-level parallelism, the model is a network of independently executing nodes each with local memory. Local memory access is very fast relative to remote access. The network protocol and topology are unimportant. Network-level parallelism is inherently scalable to as many nodes as one can afford. Optimizing to minimize the impact of network latency and bandwidth limitations is important.

For thread-level parallelism, the model is a set of threads executing on homogeneous cores that all have access to a large shared memory. The threads may outnumber the cores, in which case a multithreaded processor or multithreading operating system will timeshare the cores among the threads. The memory may be virtually and physically shared, but there may be locality considerations that encourage partitioning the cores into groups, where each group has faster access to a subset or subdomain of the memory. The threads may then be partitioned into a two-level hierarchy of thread groups and threads, with each thread group assigned to a single core group. Communication and synchronization between threads in a single core group will be faster than between threads in different core groups. If data can be partitioned between the memory subdomains and work partitioned across the thread groups assigned to the appropriate core group, performance can be further enhanced.

For SIMD parallelism, the model is synchronous SIMD or vector operations in a single thread, optimized when the data access pattern corresponds to consecutive memory locations. Other data access patterns (strided, indexed) can be supported, but with a performance penalty.

Accelerators appear in this machine model as another set of cores or core groups that are different from the main CPU cores. Work must be explicitly targeted at the accelerator, either manually by the programmer or automatically by the compiler or other tool. Each accelerator may have its own memory, shared among the accelerator cores, but distinct from the memory of the CPU. Programming the accelerator is very like programming the CPU on the node, with thread and SIMD parallelism.

Currently, the most common compute accelerator is a GPU. A GPU has a number of GPU cores organized into processing engines. Each GPU thread of execution (CUDA thread or OpenCL work item) runs on a single GPU core. However, a GPU core does not execute in the same way as a CPU core. To minimize the number of gates used for instruction fetch and dispatch and to take advantage of the similarity of code executed by graphics threads, GPU thread execution is optimized for the case where groups of adjacent GPU threads fetch and execute the same instruction

at the same time. The NVIDIA term for this group is a *warp*, the AMD term is a *wavefront*. If all the threads in a *warp* or *wavefront* follow the same execution path, the GPU runs at full efficiency. If threads take different paths (a different conditional branch), efficiency suffers correspondingly. NVIDIA calls this single-instruction, multiple-thread (SIMT) execution. From the algorithm designer's viewpoint, optimizing performance for this execution style is essentially the same as optimizing for SIMD execution: consecutive loop iterations executing the same code path simultaneously and operating on adjacent data locations. The different *warps* or *wavefronts* on a single processing engine or on different engines execute more like different CPU threads, where a branch in one warp has no effect on the performance of another warp. So the same multilevel parallelism hierarchy applies to GPUs: SIMD parallelism, implemented by the threads within a warp or wavefront, thread-level parallelism, implemented at the warp or wavefront granularity, and thread-group parallelism, implemented by the warps in all the CUDA thread blocks or OpenCL workgroups.

Other accelerators have been used and certainly new accelerators will be developed in the future. However, there are only a limited number of ways to improve performance with a new chip: more work per clock (parallelism), faster clocks, and avoiding stalls, where the most scalable of these is parallelism. Parallelism can be done at the microarchitectural level (multiple operations per clock, pipelining), or multiple cores (or equivalent), or SIMD/vector parallelism. Previous accelerators have used different combinations of these. For instance, the Clearspeed[5] processor had a two-core processor with wide SIMD operations, and the IBM PowerXCell 8i[6] had eight synergistic processing elements (SPEs), each with SIMD instructions. New accelerators with any hope of benefiting from performance portability will have some variation of multiple processing elements or cores with SIMD operations, and will map into the same machine model.

P^3 Programming Model

Now I propose a P^3 programming model for expressing parallelism on today's machines. This programming model depends on compilers and tools that are not yet widely available, but there is evidence that they can be implemented. The only barrier is the will to do so.

Network-level parallelism is addressed mostly with MPI today, and this will remain dominant for some time. MPI programming is quite mature, and gives the basics to allow programmers to deal with work distribution, data communication, and synchronization across the nodes of the network. Other approaches being used include partitioned global address space languages, like coarrays in Fortran, remote direct memory access (RDMA) approaches, like SHMEM,[7] and tasking runtimes, like Legion.[8] These approaches have had some success, and may become more popular and even dominant over time. Any of these approaches can be a P^3 solution for network parallelism.

Shared-memory thread-level parallelism has generated an incredible number of programming model solutions. Such parallelism has even been introduced into standard languages, such as C++ parallel algorithms and Fortran *do concurrent*, which we refer to as *stdpar*. This raises the very important question: If the programming language itself can express parallelism, why do we not use that? If the programming language parallelism is missing something, how do we augment it, improve it, or work around the limitation? The question is not whether to use OpenMP or Kokkos or OpenACC; it is why not use C++ parallel algorithms? Why not use *do concurrent*? We argue that we should move to stdpar programming, and we as a community should push our vendors and tools builders to provide good implementations of stdpar. The NVIDIA HPC compilers now support stdpar, including an option to offload the parallel operations to an attached GPU. Our experience with this has been very positive, and user feedback has confirmed that this is the right direction to go.

Early adopters of our stdpar implementation have been pleased to get the parallelism and performance with no language extensions and no directives. A recent example is STLBM, a double precision Lattice Boltzmann Method using C++ parallel STL templates.[9] The stdpar code is 11 times faster on an NVIDIA Ampere A100 GPU than all 48 cores of an Intel Xeon Gold 6240R, with no code changes, and is within 15% of the corresponding optimized CUDA implementation.

STDPAR Missing Features

We have noticed several features that the stdpar languages lack. Here we discuss some missing aspects, and how we are addressing these in our stdpar implementation.

One common pattern that is not available in either language is a parallel loop with one or more integrated reductions. Today, the programmer must save the reduction values to a separate vector or array and reduce those values in a separate operation. We are investigating how to best address this, and are proposing a *reduce* clause for the Fortran *do concurrent* construct, similar to the *reduction* clause in OpenMP and OpenACC.

Another thing missing in stdpar is control of how to exploit that parallelism. Which loop should be run in parallel across threads, or run using SIMD instructions, or collapsed? As mentioned, OpenMP and Kokkos and RAJA give the programmer that level of control, in fact require the programmer to specify that detail. In contrast, we have used the functionality in our OpenACC implementation, which can automatically determine how best to exploit many parallel constructs. OpenACC separates parallelism identification from parallelism exploitation. The OpenACC *acc parallel loop* construct actually does say that the loop is data independent, and in the absence of other clauses, the compiler chooses whether to execute the loop iterations across thread groups or threads or in SIMD mode, or all three. Identifying parallel loops is entirely target-independent, whereas choosing how to execute those loops may depend on the body of the loop, the context in which the loop appears, and the target architecture. We use that decision process developed for OpenACC in stdpar loops. Other implementations that have experience with OpenACC will have an advantage here.

There are four missing features dealing with accelerators in particular. One is control of whether to offload a parallel construct to an accelerator. Our implementation chooses to offload all parallel constructs to the accelerator. This is not always an optimal decision. Some constructs are parallel, but because of low parallelism or data locality are best left on the multicore CPU. Other constructs are serial, but again because of data locality are best offloaded to the accelerator. We are looking at ways of annotating regions, either with directives or with naming conventions, to give some control over offload decisions.

A second missing accelerator feature is control of data placement. OpenMP and OpenACC manage copies of host data on the accelerator, and other models have similar features. Our stdpar implementation depends on changing all dynamically allocated data to use CUDA Unified Memory, which lets the NVIDIA CUDA GPU driver automatically migrate pages between system and accelerator memory as needed. This has some limitations, such as capturing all the allocation sites and not being able to refer to stack frame memory on the accelerator, but the performance is generally quite good. All GPU vendors are promising true unified memory in the future, so we expect this to be a minor problem at that point. As users, we should require unified memory from any accelerator vendor, while allowing optional control over memory placement and migration or replication.

A third missing accelerator feature is control of asynchronous accelerator computation, allowing the multicore CPU thread to run ahead after launching accelerator operations. Today, our stdpar implementation runs the parallel offloaded operations synchronously. What we really want is some sort of tasking mechanism on the node, with optimized control for accelerator tasks. Many approaches have been designed for tasking, such as OpenMP inline tasks and Cilk task functions, and hopefully one or more of them will achieve standardization.

A final missing accelerator feature is control of multiple accelerators on the node. Since stdpar does not have any concept of accelerators, it certainly has no concept of multiple accelerators. Currently, most programs using multiple GPUs on a single node launch multiple MPI ranks per node, with one GPU per MPI rank. Our stdpar implementation depends on this, and requires that each CPU thread use an API call to select a single GPU as its accelerator. Our experience with trying to manage multiple GPUs from a single thread has found it to be a significant challenge both to get it working correctly and to get it performing well.

CONCLUSION

Productive performance portability is likely to require a two-level programming model, one to address network-level parallelism and another for parallelism on the node. Network-level parallelism is likely to be dominated by MPI for some time, but other approaches are being aggressively developed and have strong supporters. For node-level parallelism, the question is not which of OpenMP or Kokkos or OpenACC or OneAPI[10] (itself based on SYCL[11]) to adopt. That is why stdpar is not used, standard language parallelism. The NVIDIA HPC compilers already support this in C++ and Fortran, including for GPU accelerators, providing evidence that it can be an effective and efficient path for parallel programming. Newer, higher productivity languages (such as Python or Julia) that support the scalable parallelism similar to that in C++ and Fortran, either natively or with extensions, should be no more of a challenge to implement and optimize. This requires some intelligence in the compiler to offload work from the programmer to the tool. But offloading work from the programmer is exactly the job of a good compiler. For the programmer, less programming is more productive.

There will always be some regions of a program that require lower level optimizations, the same way that some routines are written in machine code today. This means that any stdpar implementation must be fully interoperable with other parallel code strategies,

which the NVIDIA HPC compilers have worked hard to achieve. There are some performance features that are not available in stdpar today. This means we have to either work to enhance the standard languages, or work harder with the system vendors and in the compilers to support them. It is work, but work done at the system hardware and software level improves the performance, portability, and productivity of all application programmers.

REFERENCES

1. H. C. Edwards and C. R. Trott, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *J. Parallel Distrib. Comput.*, vol. 74, no. 12, pp. 3202–3216, 2014.
2. D. A. Beckingsale *et al.*, "RAJA: Portable performance for large-scale scientific applications," *Proc. IEEE/ACM Int. Workshop Perform., Portability Product. HPC*, Denver, CO, USA, Nov. 2019, pp. 71–81.
3. IBM, "Fortran automatic coding system for the IBM 704," Oct. 1956. [Online]. Available: https://archive.computerhistory.org/resources/text/Fortran/102649787.05.01.acc.pdf
4. L. Higbie, "Vectorization and conversion of fortran programs for the Cray-1 (CFT) compiler," Cray Res. Inc., Tech. Note 2240207, 1978.
5. I. N. Kozin, "Evaluation of clearspeed accelerators for HPC, science and technology facilities council," Daresbury Lab., Warrington, U.K., Tech. Rep. DL-TR-2009-001, Aug. 2009. [Online]. Available: http://epubs.stfc.ac.uk
6. K. J. Barker *et al.*, "Entering the petaflop era: The architecture and performance of roadrunner," in *Proc. ACM/IEEE Conf. Supercomput.*, Nov. 2008, pp. 1–11, doi: 10.5555/1413370.1413372.
7. B. Chapman *et al.*, "Introducing OpenSHMEM, SHMEM for the PGAS community," in *Proc. Partitioned Glob. Address Space Conf.*, Houston, TX, USA, Oct. 2010, doi: 10.1145/2020373.2020375.
8. M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: Expressing locality and independence with logical regions," in *Proc. Int. Conf. High Perform. Comput., Netw., Storage Anal.*, Nov. 2012, pp. 1–11, Art. no. 1, doi: 10.5555/2388996.2389086.
9. J. Latt, "Fluid dynamics on GPUs with C parallel algorithms: State-of-the-art performance through a hardware-agnostic approach," presented at the GPU Technol. Conf., Apr. 2021. [Online]. Available: https://www.gitlab.com/UnigeHPFS/stlbm
10. Intel Corp., "Intel oneapi DPC /C compiler developer guide and reference," Apr. 2021. [Online]. Available: https://software.intel.com/content/dam/develop/external/us/en/documents/oneapi_dpcpp_cpp_compiler.pdf
11. J. R. Hammond, M. Kinsner, and J. Brodman, "A comparative analysis of Kokkos and SYCL as heterogeneous, parallel programming models for C applications," in *Proc. Int. Workshop OpenCL*, 2019, pp. 1–2, doi: 10.1145/3318170.3318193.

MICHAEL WOLFE is with NVIDIA Corporation, Hillsboro, OR, USA. He received the Ph.D. degree from the University of Illinois at Urbana in 1982, and has worked on parallel compiler analysis and optimization for the past 45 years. He received the SC Test of Time Award in 2017 for his Supercomputing 1989 paper, "More iteration space tiling," and is the author of *High Performance Compilers for Parallel Computing* (Addison-Wesley, 1996). He is a member of the ACM. Contact him at mwolfe@nvidia.com.