TACC Technical Report IMP-10

Thoughts on models for parallelism

Victor Eijkhout*

February 22, 2016

* eijkhout@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

In this note we discuss the difference between programming model and execution model, dwelling on conflicting demands on them. We use that to argue that the Integrative Model for Parallelism hits the best of both worlds, using a simple programming model that is translated to a sophisticated execution model.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code generation (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.
IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)

1 On models

We start by disentangling the concepts of execution model and programming model underlying the activity of (parallel) programming.

1.1 Execution model

With every architecture comes an execution model: a abstract description of how execution proceeds on that architecture. Typically, an execution model is close to how the actual hardware functions, and in current hardware that is mostly the Von Neumann model of a sequence of instructions that operate on memory addresses. This Von Neumann model is practically the only model in existence; 1980s experiments with dataflow hardware have not been followed up.

1.1.1 The user-visible and the internal execution model

We note that superscalar processors with out-of-order execution do not actually conform to this model, making it truly a *model*. In fact, on a level inaccessible to the user, processors have considerable dataflow machinery built in. Therefore, the Von Neumann model is in a way a programming model as described below, with dataflow as the execution model on the deepest level.

1.1.2 Parallel execution models

In MPI parallel applications, the execution model is close to Communication Sequential Processes (CSP) [6]: a collection of sequential processes that exchange messages and change their workings accordingly.

Most shared memory systems have an execution model that can be described with *sequential consistency*: the programming is executed in parallel with semantics identical to a sequential execution.

1.2 Programming model

On the other end of the spectrum a programmer expresses an algorithm in terms of a programming model: an abstract vocabulary that serves to express algorithms, but that does not necessarily correspond closely to the execution model of the hardware. Most imperative programming languages have a programming model that is close to the Von Neumann model, and are therefore in close correspondence to the execution model. The 1980s hope that imperative language could be compiled to dataflow shows that the programming model can be very different from an execution model.

Much of the work on dataflow on a task level again conflates the programming model and the execution model. It is assumed that there is a software layer that can handle task dependencies and schedule tasks accordingly, and the programmer is given a vocabulary to express these tasks and their dependencies, most notably in some graphical programming systems where task dependencies were explicitly denoted as arcs connecting dependent tasks.

1.3 Parallelism and models

The context of parallel programming offers more examples of the difference between a programming model and an execution model. For instance, the execution model for distributed memory computing is that of two-sided messaging. Since this is also quite cumbersome to program, the history of large-scale computing is full of attempts to layer a programming model on top of it that hides the ugly reality beneath a more expressive vocabulary. While the MPI library in its original form is very close to the execution model, one-sided models (now part of MPI) are one step removed, active messages take another step away, and PGAS approaches try to make distributed memory look global to the programmer. The latter programming model is by far the easiest to write, unfortunately it is also the hardest to compile and execute efficiently.

1.4 Performance/cost model

A good programming model needs a performance model or cost model to estimate performance as a function of input and platform parameters. The performance model is approximate but essential: without it the programmer has no insight into a program's likely performance.

Refer to Gropp article in CISE

Mention LogP

2 Sequential vs parallel semantics

Scientific computing, and more recent data-centric computing, is often characterized by a form of data parallelism: activities that admit of a unified description, such as finding a cosine similarity between two vectors or doing a Finite Element grid update, but that are executed spread out over many computing elements, not necessarily tightly coupled, such as SIMD lines, threads, cores, processors, et cetera. Various attempts have been made in the last few decades to design programming systems that allow such unified descriptions (we call this 'sequential semantics') of parallel activities. The conceptual attraction to such an approach is eloquently formulated in [8]:

> [A]n HPF program may be understood (and debugged) using sequential semantics, a deterministic world that we are comfortable with. Once again, as in traditional programming, the programmer works with a single address space, treating an array as a single, monolithic object, regardless of how it may be distributed across the memories of a parallel machine. The programmer does specify data distributions, but these are at a very high level and only have the status of hints to the compiler, which is responsible for the actual data distribution[.]

A very similar concept is also argued in [9] from a point of Object-Oriented design: all coordination that is not implementing an algorithmic concern needs to be encapsulated in a parallel object.

This sort of parallelism is considerably easier to deal with than concurrency:

> [H]umans are quickly overwhelmed by concurrency and find it much more difficult to reason about concurrent than sequential code. Even careful people miss possible interleavings among even simple collections of partially ordered operations. (Sutter and Larus 2005)

2.1 Parallel semantics

On the other hand, there is considerable theory and practice based on a model of parallelism that considers the execution on individual processing elements as a task by itself, rather than part of macro task, and then describes the interaction. The reality of computer architecture argues strongly for the merits of this second approach, since it explicitly accounts for the locality of data to processing elements. In current computer architectures the cost of data motion is not to be neglected, so such a system that explicitly handles locality is a guarantee for high performance. (Prime evidence here is of course the MPI library, which is the only way to get million-way parallelism with high efficiency, and which requires the user to spell out all data movement explicitly.)

However, explicit management of locality is a large burden on the programmer, so there have been many forays into systems with sequential semantics that manage locality for the programmer.

2.2 Things that are not sequential semantics

- Sequential consistency. This says that the execution of a program should give an indentical result in pallel as if it were done sequentially. However, this is based on multiple threads making their way through the source, not a single.
- SPMD (Single Program Multiple Data). Here the same program gets executed by more than one processor. The difference is that each program has a notion of identity, and of there being other processes: there is no global description.
- BSP (Bulk Synchronous Processing) [12] uses supersteps separated by barriers. First of all, BSP is SPMD in nature, secondly, BSP is also an execution model, since the barriers exist in the execution. This is in fact *of necessity* so, since the one-sided communication misses the acknowledgement signal that would allow asynchronous execution.
- PGAS (Partitioned Global Address Space). This is actually somewhat close. However, it is a form of instruction parallelism, in that different instructions are done by different processors.

2.3 Strength of the sequential semantics model

Our model, aiming at a generalized form of data parallelism, is not as general as other parallel programming systems. Specifically, this means that certain task graphs can not result from an IMP code.

Examples: X10 [1] has the *async-finish* mechanism [7] where a number of independent threads have a joint finish. This is equivalent to our model. On the other hand, the *future async-finish*

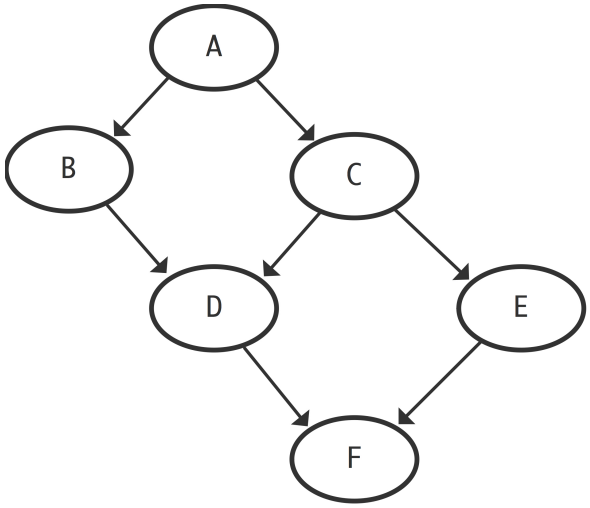

Figure 1: A task graph in the 'future async-finish' model

model (see figure 1) can generate task graphs that can not be generated by Integrative Model for Parallelism (IMP).

3 Dataflow

The Integrative Model for Parallelism (IMP) uses 'kernels' to express algorithms. These are global description of operations, formulated in global coordinates and obeying sequential semantics. We only specify data relations between kernels, not the exact sequence of execution, so this is a first example of *dataflow programming*. See for instance figure 2 for the kernel structure of an N-body algorithm.

By analysis of the distributions of work and data, this global and sequential (and therefore synchronous looking) expression is translated to a dataflow formulation. We call this an Intermediate Representation (IR), because dataflow is not directly an execution model. As we have argued elsewhere, dataflow can be considered to be a programming model that unifies multiple execution models such as message passing and task graphs. Since both of these are asynchronous (subject to user introduction of explicit synchronization of course), we have shown that the sequential semantics of the IMP programming model can correspond to an asynchronous execution model.

The *dataflow* idea has been around for a while. For instance, Dennis [2] gave a definition, and argued that it was functional in nature.

Traditionally, dataflow uses *tokens* which combine aspects of data and control. We use a different synchronization mechanism [3].

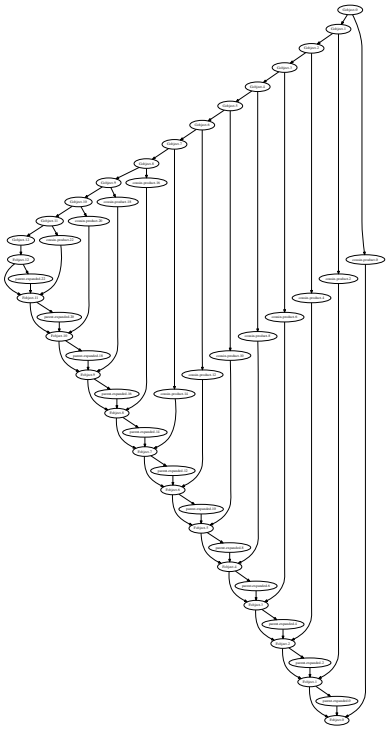

Figure 2: Relation between the kernels of an N-body algorithm

4 More thoughts

I haven't figured this out yet: one reason that IMP can make such claims is that it builds up a limited representation of the program in the program itself. Here are two concepts that may express this.

Inspector-executor One HPC system, predating MPI, is the "Parti primitives" [10]. This package originated the *inspector-executor* model, where the user would first declare a communication pattern to an inspector routine, which would then yield an object whose instantiation was the actual communication. This expresses the fact that, in High Performance Computing (HPC), communications are often repetitions of the same irregular pattern. The *inspector-executor* model is currently available in the `VecScatter` object of the PETSc library [4], and the `map` objects of Trilinos [5].

Partial evaluation and multi-stage programming Multi-stage programming [11] builds up a representation of the program in the program, and has mechanisms to then execute it.

> Having `Run` in the language is important if we want to use code constructed using the other MetaML constructs, without going outside the language.

From LtU:

> I'm not sure I understand what phenomenon you're trying to describe, but you might look into partial evaluation, adaptive optimization, dynamic recompilation, super compilation, and staged programming for some appropriate vocabulary.

References

[1] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[2] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*, pages 362–376, London, UK, UK, 1974. Springer-Verlag.

[3] Victor Eijkhout. Task execution in the integrative model. Technical Report IMP-04, Integrative Programming Lab, Texas Advanced Computing Center, The University of Texas at Austin, 2014.

[4] W. D. Gropp and B. F. Smith. Scalable, extensible, and portable numerical libraries. In *Proceedings of the Scalable Parallel Libraries Conference, IEEE 1994*, pages 87–93.

[5] Michael A. Heroux, Roscoe A. Bartlett, Vicki E. Howle, Robert J. Hoekstra, Jonathan J. Hu, Tamara G. Kolda, Richard B. Lehoucq, Kevin R. Long, Roger P. Pawlowski, Eric T. Phipps, Andrew G. Salinger, Heidi K. Thornquist, Ray S. Tuminaro, James M. Willenbring, Alan Williams, and Kendall S. Stanley. An overview of the Trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[6] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985. ISBN-10: 0131532715, ISBN-13: 978-0131532717.

[7] Jonathan K. Lee and Jens Palsberg. Featherweight x10: A core calculus for async-finish parallelism. *SIGPLAN Not.*, 45(5):25–36, January 2010.

[8] Rishiyur S. Nikhil. An overview of the parallel language id (a foundation for ph, a parallel dialect of haskell). Technical report, Digital Equipment Corporation, Cambridge Research Laboratory, 1993.

[9] Eduardo Gurgel Pinho and Francisco Heron de Carvalho Junior. An object-oriented parallel programming language for distributed-memory parallel computing platforms. *Science of Computer Programming*, 80, Part A(0):65 – 90, 2014. Special section on foundations of coordination languages and software architectures (selected papers from FOCLASA10), Special section - Brazilian Symposium on Programming Languages (SBLP 2010) and Special section on formal methods for industrial critical systems (Selected papers from FMICS11).

[10] A. Sussman, J. Saltz, R. Das, S. Gupta, D. Mavriplis, R. Ponnusamy, and K. Crowley. PARTI primitives for unstructured and block structured problems. *Computing Systems in Engineering*, 3(1-4):73–86, 1992.

[11] Walid Taha. *Multi-stage programming: Its theory and applications.* PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[12] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.