A Personal, Historical Perspective of Parallel Programming for High Performance

Christian Lengauer

Fakultät für Mathematik und Informatik
Universität Passau, Germany

lengauer@fmi.uni-passau.de
http://www.fmi.uni-passau.de/~lengauer

1 Introduction

Although work on the specification, semantics and verification of parallelism over the last decades has led to significant progress in the understanding of parallelism and to workable development methods in certain domains, no method for the development of reliable, portable, parallel application software for high-performance computing has been achieved as of yet whose practicality and ease of use is commonly evident.

Presently, the high-performance programming community is, in general, not working with a formal foundation or a rigorous discipline of programming. This may be deplored, but it is understandable when one reviews past developments in the verification and semantics of parallelism.

2 The Past

In the Seventies, verification techniques based on Hoare logic or temporal logic were introduced and immediately used with quite some success in verifying basic properties of parallel programs, like adherence of an input-output specification, or mutual exclusion, or the absence of deadlock. Abstract data types –notably, the monitor– were proposed for the administration of shared data, and were also equipped with proof rules. All this work did not explicitly address the exchange of data between processors in the form of message-passing communications.

In the late Seventies and early Eighties, distributed programming received a lot of attention, fueled by Hoare's introduction of CSP [15], in which he stressed the absence of a semantic foundation for it. There were early hopes for a straightforward extension of Owicki's shared-memory verification methods [27, 28], but the result turned out to be impractical [1, 21]. Instead, one turned to process algebra. The dominant theories were CCS [24] and a reworked version of CSP [16] and, by today, both come in many variations. These theories are convenient for the description of communication behaviour, but they cannot deal easily with the notion of an assignable local store.

One language offering message passing and a store is occam [17], which was introduced as a spinoff of CSP. However, occam failed in dominating the high performance market. One reason may be that it is just at too low a level for programming massive parallelism. We will pick this thought up in the following section.

Petri nets [30] had been around since the Sixties and had proved their worth for the specification of concurrent control. Temporal program logic had also come to fruition [22] and proved particularly useful in the specification and model checking of concurrent systems. Lately, it has been extended for a more assertional style of reasoning [19].

However, none of these foundations, which are very useful in other applications, succeeded in providing a basis for high-performance computing. One possible reason is that the granularity of parallelism of the systems they are designed for is static and comparatively low. In high-performance computing, one of the major choices to be made is the level of granularity, which depends on factors like the problem size and the machine size, and which can change frequently. High-performance programmers have a strong need for convenient aggregate data structures and flexible partitioning techniques to achieve the desired granularity.

Verification becomes a lot easier when one makes the source program look sequential. This is often called data parallelism and is particularly easy with the use of a functional language. One commendable effort on this front was Sisal [5], a functional language which competed in performance directly with Fortran. In order to win, it forewent functional pearls like polymorphism and higher-order functions. In the end, Sisal did not get the attention it was seeking: imperative programmers did not feel the need to switch and functional programmers found it difficult to accept the sacrifices it made in the interest of performance.

By the late Eighties, hardware technology had created a parallel computer market which amplified the trend towards message passing for scalable parallel programs, i.e., programs whose amount of parallelism scales up with the problem size and which become, for large problem sizes, massively parallel. The need for working high-performance programs had grown and researchers closer to applications devised language constructs which enabled them to produce high-performance software. These constructs were extensions of or could be combined with sequential core languages which the high-performance sector favours – mainly, Fortran and C. There were two major approaches.

The first approach extended the core language with constructs for parallelism and distribution of work and data. After many individual efforts, an attempt at a standardization was made in the early Nineties with the definition of High-Performance Fortran (HPF) [18]. HPF is Fortran adorned with comments which point out a parallel structure –like a parallel loop or independent pieces of code– or a distribution pattern of data aggregations (arrays) across a set of processors. The compiler can heed these comments or ignore them, if they are beyond its capabilities.

The second approach paired the core language with a library of primitives for communication and synchronization. The best known representatives are the Parallel Virtual Machine (PVM) [10], the Message-Passing Interface (MPI) [29] and Bulk-Synchronous Programming (BSP) [23]. Here, the number of processes is specified by a system parameter.

3 The Present

The major portion of today's high-performance parallel programs is written in C or Fortran, augmented with MPI primitives. If one believes that occam is too low-level to have been successful, MPI, which succeeded, must be just that much more abstract: at least, one does not have to name the communication channels explicitly. But MPI programming is still very cumbersome and error-prone. Much more thought goes into working out the parallel structure of the program than into solving the computational problem at hand. There is no sound basis for establishing correctness, the number of MPI primitives (over 200) is too large to assure its maintenance as a standard, and portability remains a problem. Yet, MPI serves the immediate need of the high-performance application sector, because it produces working programs – even though often at a high price.

A smaller part of the market is trying to work with HPF. Correctness is less of a programming issue here, since the HPF program appears sequential (cf. data parallelism), but there is the burden on the compiler to get the parallel part correct and efficient. This style of programming is much more comfortable than using MPI, and present-day HPF compilers will produce good code for some problems [31]. However, the usefulness of HPF deteriorates rapidly with increasing irregularity of the computational problems. There are efforts under way to make HPF more irregularity-resiliant.

4 The Future

Can life be made easier for parallel programmers? Is there hope for portable parallel programs? Can a formal foundation for parallel programming for high performance be found? Many research efforts are under way to answer these questions positively.

One approach is to clean up the act of MPI programming. Let me mention two trends.

The first trend disciplines the programmer in the use of communications: much like the goto was banished from sequential programming, point-to-point communications could be banished from MPI programming. Instead, one would rely only on the use of more abstract, regular communication patterns. Some are already provided by the MPI library: reduce, scan, scatter, gather, etc. Techniques of program transformation can help in identifying efficient implementations for these patterns [34] and choosing the right pattern – or the right composition of patterns – even with regard to a particular processor topology [12].

The second trend disciplines the programmer by structuring the program into blocks between which no communication is allowed to take place. In BSP, these blocks are called supersteps. Communications specified within a superstep are carried out after all computations of the superstep have terminated. The benefit of this approach is an increased simplicity of programming and cost prediction. The BSP library is an order of magnitude smaller than the MPI library, and its cost model rests on only three machine parameters! The price paid is that, like with the imposition of any structure, the programmer loses flexibility –the verdict on how serious a loss this is is still out– and that there is an inherent fuzziness in the evaluation of the machine parameters [32].

Another approach which relies heavily on a smart run-time system is Cilk, Leiserson's extension of C [9]. Cilk is C plus half a dozen constructs for the definition and containment of parallelism. Leiserson addresses the issue of correctness by running systematic tests administered by a so-called nondeterminator [8]. For safety-critical applications, one should apply a method of deriving correct Cilk programs.

In the same vein, Glasgow Parallel Haskell (GpH) extends the functional language Haskell [33]. Here, the effects of the extensions on the language semantics are easier to understand: the program's output values cannot be corrupted, only the time of their appearance can change (albeit, maybe, to infinity)!

Functional programs are particularly amenable to program transformation, also for exposing parallelism. In the last decade, people have tried to identify computational patterns which contain inherent parallelism and to derive this parallelism – in the best case, systematically through a sequence of equational program transformations. The aim is to form a library of these patterns, backed up with efficient implementations for existing parallel computers. Systems are being worked on which can make use of such libraries [2, 6]. There is also some industrial involvement [3], but it is still in its infancy.

Particular attention has been paid to the divide-and-conquer paradigm, which comes in a variety of patterns [11, 13, 26]. There are also functional languages specifically supporting divide-and-conquer [4, 14, 25].

The use of patterns –or schemata, skeletons, templates, or whatever one might call them– is also an advance in that it is highly compositional. This benefit, often obtained when one imposes structure, does unfortunately not apply to the aspect of performance: in general, the composition of two patterns, which have been tuned individually for a given architecture, must be retuned [12, 34].

The most powerful way of unburdening the programmer from correctness and performance issues of parallelism is, of course, to use certified automatic methods of parallelization. In the imperative world, much effort has been invested in the automatic parallelization of nested loops. This is the focus of every parallelizing compiler. The polytope model for loop parallelization [7, 20], which emerged from systolic design, goes much further than present-day parallelizing compilers by providing an optimizing search for the best parallel implementation. But obtaining a solution in a model is much easier than deriving competitive target code. Much work remains to be done and is being done here.

Let me conclude with the observation of two fashionable trends of today.

The first is the increasing interest in the new library OpenMP (openmp.org), the shared-memory equivalent of MPI, which is pushed heavily by an industrial consortium. There is the hope that virtual shared memory will make any consideration of the location of a datum in a distributed store obsolete. With the increased use of clusters of symmetric shared-memory processors (SMPs), a combination of MPI and OpenMP programming will likely prevail in the near future.

The second is a large impetus concerning Java for high-performance computing, which has led to the formation of an interest group, the Java Grande Forum (javagrande.org). As long as Java is interpreted by a virtual machine, it will be difficult to obtain high performance – but this will not last forever. There are other aspects of Java which make it seem like an unlikely candidate for high-performance computing: its thread model is inefficient, the array –the data structure used most for high performance– cannot be given multiple dimensions easily and there are problems with its floating-point arithmetic. However, the pressure for an adaptation of Java to high-performance computing in some form or other is mounting rapidly, with many researchers getting involved.

5 Conclusions

It seems quite clear that structureless parallelism is not going to have a future in high-performance computing – be it with shared memory or with message passing. The step yet to be taken in a major part of the applications is akin to the switch from assembly programming to higher-level languages in the Sixties – a step of abstraction. Just as back then, it will require advances in language design, compilation technology and parallel architectures, and it is still going to be painful for programmers, who will feel they give up essential liberties.

In high-performance computing, parallelism is not part of the specification – a performance requirement is! Parallelism enters the stage only in the program development, as an optimization to attain the required peformance. Consequently, one has a choice of how to structure one's parallelism. This structure is more easily imposed when the computation has structure itself than when the structure of the computation is irregular or unpredictable. Especially in the latter case, optimal performance is often going to be difficult to achieve.

Freeing the programmer completely from the issue of parallelism is not going to be practical, except in trivial cases. Even if the compiler chooses the parallel structure based on knowledge of the target machine and the dependences in the program, the programmer is still the maker of these dependences and, thus, predetermines the potential for parallelism. Granted: present-day parallelizing compilers alter the dependence structure specified by the programmer –e.g., by expanding scalars to vectors or by plugging in parallel code for reductions and scans– but, for alterations at a larger scale, the use of custom-implemented patterns seems more promising. (Note that reductions and scans are basic patterns themselves.) In general, it is going to be a good idea to let the programmer assist the compiler with simple hints.

The present phase in the development of high-performance computing is one of exploration. So, by the way, is the present phase of architecture design. In the quest for high performance, computer architectures are becoming more ornate and, as a consequence, are increasing the burden on high-performance compilers. Whether this tendency is going to be sustained and compilers can keep up with it, remains to be seen.

To me, the most dominant issue in high-performance computing, which has hardly been addressed effectively so far, is that of portable performance. After all, a high-performance computer usually becomes obsolete after about five years!

6 Disclaimer

This treatise is meant as a quick, to-date, personal perspective. I do not claim objectiveness or completeness, and I am aware of the personal bias in my citations. On the other hand, it has not been my intention to omit anything on purpose or offend anyone.

7 Acknowledgements

My sincere thanks go to the members of my group: Nils Ellmenreich, Peter Faber, Sergei Gorlatch, Martin Griebl and Christoph A. Herrmann. My opinions are being tested and adjusted in my daily work with them. Thanks to Jeff W. Sanders for reading and discussing of an early draft.

References

1. K. R. Apt, N. Francez, and W. P. de Roever. A proof system for communicating sequential processes. *ACM Trans. on Programming Languages and Systems*, 2(3):359–385, July 1980.
2. B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: A structured high level programming language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, 1995.
3. B. Bacci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Skeletons and transformations in an integrated parallel programming environment. In *Parallel Computing Technologies (PaCT-99)*, LNCS 1662, pages 13–27. Springer-Verlag, 1999.
4. G. E. Blelloch, J. C. Hardwick, J. Sipelstein, M. Zagha, and S. Chatterjee. Implementation of a portable nested data-parallel language. *J. Parallel and Distributed Computing*, 21(1):4–14, Apr. 1994.
5. D. Cann. Retire Fortran? A debate rekindled. *Comm. ACM*, 35(8), Aug. 1992.
6. J. Darlington, A. Field, P. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. L. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *Parallel Architectures and Languages Europe (PARLE '93)*, LNCS 694, pages 146–160. Springer-Verlag, 1993.
7. P. Feautrier. Automatic parallelization in the polytope model. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, LNCS 1132, pages 79–103. Springer-Verlag, 1996.
8. M. Feng and C. E. Leiserson. Efficient detection of determinacy races in Cilk programs. In *9th ACM Symp. on Parallel Algorithms and Architectures (SPAA'97)*, pages 1–11. ACM Press, 1997.

9. M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM SIGPLAN Notices*, 33(5):213–228, May 1998. *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI'98)*.
10. A. Geist et al. *PVM: Parallel Virtual Machine*. MIT Press, 1994.
11. S. Gorlatch. Abstraction and performance in the design of parallel programs. Technical Report MIP-9802, Fakultät für Mathematik und Informatik, Universität Passau, Jan. 1998. Habilitation thesis.
12. S. Gorlatch, C. Wedler, and C. Lengauer. Optimization rules for programming with collective operations. In *Proc. 13th Int. Parallel Processing Symp. & 10th Symp. on Parallel and Distributed Processing (IPPS/SPDP'99)*, pages 492–499. IEEE Computer Society Press, 1999.
13. C. A. Herrmann and C. Lengauer. Parallelization of divide-and-conquer by translation to nested loops. *J. Functional Programming*, 9(3):279–310, May 1999.
14. C. A. Herrmann, C. Lengauer, R. Günz, J. Laitenberger, and C. Schaller. A compiler for HDC. Technical Report MIP-9907, Fakultät für Mathematik und Informatik, Universität Passau, May 1999.
15. C. A. R. Hoare. Communicating sequential processes. *Comm. ACM*, 21(8):666–677, Aug. 1978.
16. C. A. R. Hoare. *Communicating Sequential Processes*. Series in Computer Science. Prentice-Hall Int., 1985.
17. INMOS Ltd. occam *Programming Manual*. Series in Computer Science. Prentice-Hall Int., 1984.
18. C. H. Koelbel, D. B. Loveman, R. S. Schreiber, G. L. Steele, Jr., and M. E. Zosel. *The High Performance Fortran Handbook*. Scientific and Engineering Computation. MIT Press, 1994.
19. L. Lamport. TLA—temporal logic of actions. *ACM Trans. on Programming Languages and Systems*, 16(3):872–923, May 1994.
20. C. Lengauer. Loop parallelization in the polytope model. In E. Best, editor, *CONCUR'93*, LNCS 715, pages 398–416. Springer-Verlag, 1993.
21. G. M. Levin and D. Gries. A proof technique for communicating sequential processes. *Acta Informatica*, 15(3):281–302, 1981.
22. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.
23. W. F. McColl. Scalable computing. In J. van Leeuwen, editor, *Computer Science Today*, LNCS 1000, pages 46–61. Springer-Verlag, 1995.
24. R. Milner. *Communication and Concurrency*. Series in Computer Science. Prentice-Hall Int., 1989.
25. Z. G. Mou. Divacon: A parallel language for scientific computing based on divide-and-conquer. In *Proc. 3rd Symp. Frontiers of Massively Parallel Computation*, pages 451–461. IEEE Computer Society Press, Oct. 1990.
26. Z. G. Mou and P. Hudak. An algebraic model for divide-and-conquer algorithms and its parallelism. *J. Supercomputing*, 2(3):257–278, 1988.
27. S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.
28. S. S. Owicki and D. Gries. Verifying properties of parallel programs. *Comm. ACM*, 19(5):279–285, May 1976.
29. P. S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann, 1997.
30. W. Reisig. *A Primer in Petri Net Design*. Springer Compass International. Springer-Verlag, 1992.

31. R. Schreiber. High Performance Fortran, Version 2. *Parallel Processing Letters*, 7(4):437–449, 1997.
32. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
33. P. W. Trinder, K. Hammond, H.-W. Loidl, and S. L. Peyton Jones. Algorithm + strategy = parallelism. *J. Functional Programming*, 8(1):23–60, Jan. 1998.
34. C. Wedler and C. Lengauer. On linear list recursion in parallel. *Acta Informatica*, 35(10):875–909, 1998.