What is Easy?

E.F. Van De Velde

CRPC-TR90046
March, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

What is easy? *

Eric F. Van de Velde
Applied Mathematics 217-50
Caltech
Pasadena, CA 91125

CRPC-90-1
March 7, 1990

Abstract

According to our new Center poster "The goal of the Center for Research on Parallel Computation is to make parallel computers usable for high-performance scientific computing problems." In this CRPC forum, I will lead a discussion on the fundamental aspects of this goal: Why is concurrent programming difficult? How can concurrency be simplified? What is usable? What is easy?

To kick off the discussion, I shall present one view and think through its consequences by means of a specific case study. Hopefully, this will inspire a debate on the design of concurrent scientific programs.

1 Introduction

Concurrent programming is difficult and needs to be simplified. This simple statement is the most important goal of the Center for Research in Parallel Computation. The new focus on simplification is a natural extension of early research in concurrent computing, which was concerned mainly with feasibility. The accumulated experience of feasibility studies is overwhelmingly positive. For almost all concurrent machines that were available for an extended period, a substantial number of core applications were implemented, and all but a few exceptions ran efficiently. These feasibility studies

*Subject of a CRPC discussion forum. This material is based upon work supported by the NSF under Cooperative Agreement No. CCR-8809615. The government has certain rights in this material.

required machine-dependent program and problem reformulation. To raise the concurrent technology from the level of feasible to that of usable, current research focuses on simplification of the concurrent-programming task. Although an easily stated goal, a solid interpretation lacks: When is one computing system easier than another? What makes a program easy or difficult? What does "this program is easy to parallelize" mean? What is easy?

These questions are rarely asked or answered explicitly. Instead, we are guided by some vague and intuitive notions, typically shaped by (senior) colleagues or derived from the computer systems we most frequently used. With significant practical experience available, we must now examine these fundamental questions more carefully. A good answer will allow us to approach the task of simplification more scientifically, because it will point to objective criteria for evaluating how difficult concurrent systems are. What may result is a solid framework for the heuristics that have been and are being developed.

The concurrent-programming task involves several levels: architectures, languages, compilers, software tools, and applications. Before addressing the main question, I shall examine complexity and/or simplicity at each level. After this overview, I shall give a plausible definition of the term "easy." In the mean time, its intuitive meaning will suffice. The term efficiency is used informally throughout the text, although it is assumed that reasonable measures of efficiency exist.

2 Architectures

The two reasons for concurrency are hardware-related:

- Concurrency avoids the fundamental limit on the sequential computing speed, ultimately determined by fundamental laws of physics.
- Systems that consist of many duplicated components are inherently more economical.

There are no other inherent benefits to concurrency: anything that can be computed concurrently in finite time and with a finite number of processors can be computed sequentially in finite time.

Historically, computer architectures have grown ever more complex. What is different about the evolution toward concurrent machines, is that the end user must be aware of the concurrency in the system. Process scheduling,

memory management, synchronization, and communication strategies based strictly on information available at the hardware and system-software level cannot meet the required efficiencies, because efficiency on concurrent computers is a global property, encompassing the whole program, including all software levels, as well as the hardware.

Here, we must briefly expand on the importance of efficiency. If breaking fundamental computing speed barriers is the main reason for using concurrent computers (a view appropriate for national laboratories), efficiency may be so critically important that other goals, like easiness, must yield. It is not that easiness is not important; only that all techniques that lead to higher absolute performance will be used, easy or not. However, the success of concurrency according to our other view — that concurrency is a more economical way to performance, perhaps at levels that are reachable sequentially — critically relies on concurrency being easy. Efficiency is nevertheless important: a minimum requirement is that the concurrent hardware costs less than the sequential alternative with the same sustained performance level.

Whatever view of concurrency one subscribes to, both efficiency and easiness are important, albeit with different emphasis. It is tempting to express our goal in optimization terminology as follows: we wish to maximize easiness under the constraint that efficiency must be above a certain minimum level. The required minimum efficiency guarantees that the concurrent hardware is either faster or more economical than its competition. Henceforth, I shall consider only those programs that satisfy the efficiency constraint: we do not design a system to make inefficient programs easy to write. Rather, we wish to simplify writing efficient software. (This argument also holds if one replaces efficiency with correctness: it is not our intention to simplify writing incorrect programs.)

3 Languages

A computer language achieves two objectives: abstraction and notation.

Abstraction allows one to formulate problems at a level nearer to the application and more removed from the machine. Is an abstract program necessarily easier? One can definitely argue convincingly by comparing programs for clean text-book problems. It is more difficult to make the case for real-world problems that do not readily fit into an abstract framework. Abstract versions may have other desirable properties, however, like formal

verification and more general applicability.

The efficiency of highly abstract languages (determined by execution times of compiled programs) has been a problem. Is this because not enough time was spent to develop the compilers? Or must performance be poorer in principle? This debate is not within the scope of this discussion. However, one aspect of the conflict between abstraction and efficiency does seem true in principle: whenever low-level optimizations are needed, abstraction stands in the way, because the purpose of abstraction is to shield programmers from the lower levels, not to give access to them. When comparing programs of similar performance, a low-level program may be judged easier than a high-level program, because the optimization of the latter was more difficult.

Notation is another aspect of computer languages, much less ambitious than abstraction. Notation determines how the ideas embodied in a program look on paper. Although good notation is important, it is ultimately misleading as a criterion for level of difficulty. How often does one hear statements along the following lines: "Look how easy this language is; it only took ten lines of code to write this operation." Another: "We only had to change three lines to parallelize the code." At most, these criteria indicate the level of difficulty of typing the program, not of developing it. The intellectual effort goes into deciding which constructs are needed; not in writing them down.

4 Compilers

It is the task of compilers to translate source code into correct and efficient executable code (we do not consider the speed of the compilation itself). There is no compromise possible where correctness is concerned. All other goals, like efficiency, must yield. Ideally, the produced object code runs at a speed close to predicted by operational counts and machine performance. To achieve this on supercomputers, interaction between user and compiler is commonplace, because compilers cannot perform automatically all task scheduling and data-dependency analyses necessary to obtain efficiently running programs. The interactions take the form of compiler directives or program restructuring, often suggested by the compiler. Because barely readable restructured code and machine specific compiler directives that implement low-level details add an extra layer of complexity, writers of parallelizing compilers now put an increased emphasis on the design of the user interface.

Software tools for performance analysis and visualization help the user to locate bottlenecks and other performance troubles. An inherent limitation is that these tools are diagnostic only, although some problem solving capability is obtained if performance analysis tools and parallelizing compilers are combined into one expert system. The development of techniques that avoid performance troubles in the first place are inherently preferred (whether such techniques could be packaged as software tools is unlikely).

When evaluating particular computing systems, we must consider all steps necessary to produce code that meets all requirements, including efficiency.

5 Software

Software is the last level that may hide problems left unresolved by either architecture, language, or compiler. The main issue in concurrent software development is the definition of interfaces. Experience shows that it is difficult to combine two independently developed software packages for concurrent computers into one application. E.g., two incompatible protocols for communication might have been used. In this case, one must rewrite the communication aspects of at least one of the packages.

6 Applications

Today, concurrency is merely a nuisance for application programmers. Only researchers interested in the concurrent computing aspect itself put up with it, hoping for a future dividend in performance or economy.

Application programmers typically have several alternatives for solving their computing problems. This freedom of choice can be exploited to tailor the application to the computer. New numerical methods that have a higher degree of concurrency can only be welcomed. The fate of concurrent computing would be disastrous, however, if its success hinged on the invention of new numerical techniques for every new architecture. The judgment of the success or failure of concurrent computing should be based on the (un)availability and efficiency of classical methods.

7 What is easy?

Let us imagine a future of concurrent computing. The research in the areas described above has blossomed, and the creative genius of many researchers has delivered a wide variety of architectures, languages, compilers, and an array of software tools. Our future programmer sits in front of his or her ultra-high-resolution-graphics work station (the ConNeXT) with a color pallet that rivals Da Vinci's, HI-FI sound, and built-in voice recognition. Of course, the system is highly concurrent.

Our programmer, a budding genius who wishes to test his/her new theory for turbulence, is ready to start dictating his or her program. A possible scenario of the programming effort follows:

1. Invent a new concurrent method to solve the Navier-Stokes equations.
2. Write program in high level language.
3. Use fast compiler, one that produces inefficient code, during initial stages of code development until code satisfies formal specifications.
4. Iterate the following steps:
   (a) Use parallelizing compiler in an effort to meet performance specifications.
   (b) Introduce compiler directives, telling the compiler how to compile loops, distribute data, vectorize, assemble data into messages, etc.,... in an effort to increase performance even more.
   (c) Use feedback from the compiler-expert system to restructure program. (If an error creeps in while restructuring, start over.)
   (d) Use performance evaluation tool to identify bottlenecks and other performance problems.

Is this the future we want?

(Concurrent) programming is difficult because many goals must be met simultaneously: formal correctness, load balance, minimum communication, synchronization, vectorization,... Each one of these goals must be weighed against all others, because solving one problem is usually at the expense of another. Concurrency is not one difficult issue; it is a set of mutually interfering issues, each of which — when considered separately — is quite manageable. Many small problems interfering with one another, that requires

strategic thinking, that requires the iterative process of code optimization, that requires user input.

If we accept current wisdom that concurrency requires human intervention, what is the best one can hope for? That, for the user, concurrency is reduced to one single issue. The hardware and all system software should be geared toward eliminating all other issues, toward removing the interdependencies, and toward simplifying the one issue that needs to be resolved by the user.

What is easy? One single, independent problem.

According to this criterion, we make concurrency easy by reducing it to one single issue. If realized, the need for strategy in concurrent code development is reduced dramatically, because there are no mutually conflicting requirements to be balanced. Strategy, if any is required, is limited to optimizing the one remaining issue.

8 A case study

We choose one particular issue for the user to resolve (let us call it the primary issue). All others are relegated to the underlying levels of hardware, language, compiler, and software. The implementation of such a global view of concurrency requires a multi- and interdisciplinary approach, impossible within the scope of a limited one-person project. It was possible, however, to develop one particular concurrent software package according to our simplicity criterion. We chose the narrow application area of linear algebra as a test bed. In spite of the restriction that all ideas had to be implemented at the software level, excellent performance was obtained. Moreover, to obtain the best performance for a particular program using this package, the user needs to resolve the primary issue only. By this criterion, this particular concurrent software package is easy. We stress, however, that the application of our answer to "What is easy?" transcends the narrow scope of these initial implementations.

We chose data distribution as the primary issue. This is not the only possible choice, perhaps not the best choice. Its justification is that we were able to develop easy software, in the sense that the introduction of a data distribution is sufficient to obtain an efficient concurrent program from a sequential one. The user assigns a locality to all values occurring in

the execution of a program. In a local memory environment, a value must be assigned to a variable of a particular process. Processes are assigned to particular processors (in this framework, a process is a value and a processor is a location). The mapping of values to locations may, in principle, be dynamic. The mapping of values to processes determines the amount of work done by each process. Both mappings have a bearing on the load balance and on the communication.

For the implementation of our linear algebra package, we identify processes by means of a process coordinate pair (p,q), with $0 \leq p < P$ and $0 \leq q < Q$. Each matrix entry $a_{m,n}$ of an $M \times N$ matrix A is allocated to a particular process by means of the mappings $p = \mu(m)$ and $q = \nu(n)$. Vectors are distributed compatibly with either the row distribution or the column distribution of the matrix. We consider the first only, the other follows by duality. With $p = \mu(m)$, entry v_m of the M-vector $\vec{v}$ is mapped to all processes with first coordinate equal to p. Hence, the entry is duplicated Q times. Scalars are duplicated in every process.

What are the consequences of this approach for developing the software? The most far reaching is that all component subroutines must be written so that they are independent of the data distribution. This is trivially possible for most matrix-vector operations. For LU-decomposition, it required implicit instead of explicit pivoting, see [2]. For QR-decomposition, it required a deeper understanding of the connection of this algorithm with recursive doubling, see [3].

According to our easiness-criterion, this software package is easy if an efficient user program is obtained purely by tuning the data distribution. Thus, the user's sole task (as far as concurrency is concerned) is to supply the mappings μ and ν. Can a good choice substantially affect the efficiency? Is the best efficiency thus obtained close to optimal? In tests, both questions are answered positively. Although some low-level efficiencies cannot be exploited by the software, other high-level efficiencies become accessible, more than offsetting any losses.

Low-level inefficiencies that were incurred include the need for memory access in irregular patterns and the impossibility of exploiting properties of particular data distributions. E.g., Geist and Romine in [1] exploit the combination of a particular data distribution and pivoting strategy to overlap some computation and communication in LU-decomposition. Because in our approach the data distribution is unknown, such strategies are inaccessible.

Consider the following simple example to clarify the type of high-level efficiencies that are accessible in our approach. Let A be an $M \times N$ matrix,

and $\mathbf{x}$ and $\mathbf{y}$ vectors of dimension N and M, respectively. The implementation of the assignment:

$$\mathbf{y} := A\mathbf{x} \tag{1}$$

requires the evaluation of a matrix-vector product. If this were a self-contained program, not part of a larger program, the optimal data distribution and corresponding optimal program is easily derived. For a computation with P concurrent processes, one should:

- Distribute A by rows only, i.e., $Q = 1$.
- Allocate M/P rows of A to each process.
- Distribute $\mathbf{y}$ compatibly with the row distribution of A.
- Duplicate the vector $\mathbf{x}$ in each process.

The resulting program is optimal, because it is perfectly load balanced (if M is divisible by P) and it requires no communication. Similarly, for the assignment:

$$\mathbf{z}^T := \mathbf{y}^T A \tag{2}$$

one should apply a data distribution dual to the one for (1), i.e., reverse the role of rows and columns.

For a composite program that evaluates both assignments (1) and (2) neither distribution is optimal. The best distribution is a function of the ratio of the number of times (1) versus (2) is evaluated. Generally, the best distribution has both $P \neq 1$ and $Q \neq 1$. If the software package were to supply these operations for their respective "optimal" distributions only, a data redistribution is necessary between evaluations of (1) and (2).

In [4], a more realistic example is given. There, the data distribution is dynamically adapted to the pivot locations of LU-decompositions. In this way, maximum load balance is obtained throughout the computation. This achieves vectorization of the code as well. The data distribution is the only concurrency-related issue addressed at the user level. All code-optimization strategy is limited to adapting the data distribution.

These examples show that optimality is not preserved under program composition. This has the important practical consequence that performance often decreases as a function of program size. A global approach toward concurrent program optimization has a higher probability of avoiding this problem than the loop by loop approach, which inherently does not

address the global nature of concurrent performance. The examples show how global optimization might work.

For our approach to be generally applicable, the inherent limitation of working at the software level only must be removed. If data distribution were adopted as the primary issue, one could develop data-distribution-independent languages and/or compilers, methods for dynamically changing data distributions, node hardware that makes it easier and more efficient to access memory in irregular patterns (e.g., scatter/gather hardware),...

9 Conclusion

Interdisciplinary collaboration has long been recognized as crucial to make concurrency work. A problem of such collaborations is to combine in a homogeneous and consistent framework the many ideas coming from different directions. Currently, concurrency is a hodgepodge of techniques based on heuristics. To streamline, we must have a common goal. My answer to "What is easy?" is an attempt to provide such a goal: the elimination of all but one concurrency-related issues.

References

[1] G. A. Geist and C. H. Romine. LU factorization algorithms on distributed-memory multiprocessor architectures. *SIAM Journal on Scientific and Statistical Computing*, 9(4):639–649, July 1988.

[2] E. F. Van de Velde. *Experiments with Multicomputer LU-Decomposition.* report CRPC-89-1, Center for Research in Parallel Computing, 1989. To appear in Concurrency: Practice and Experience.

[3] E. F. Van de Velde. Multicomputer matrix computations: theory and practice. March 1989. Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications.

[4] E. F. Van de Velde and J.L. Lorenz. *Adaptive Data Distribution for Concurrent Continuation.* report CRPC-89-4, Center for Research in Parallel Computing, 1989.