Program Speedups in Theory and Practice

Neil D. Jones[a*]

[a]Department of Computer Science, University of Copenhagen, Universitetsparken 1, DK-2100 Copenhagen, Denmark

Abstract

The aim of this discussion paper is to stimulate (or perhaps to provoke) stronger interactions among theoreticians and practitioners interested in efficient problem solutions. We emphasize computing abilities in relation to *programming and specification languages*, as these provide the user's interface with the computer.

Due to the breadth of the topic we mainly discuss efficiency as measured by *computation time*, ignoring space and other measures; and this on *sequential, deterministic, individual, and fixed computers*, not treating parallelism, stochastic algorithms, distributed computing, or dynamically expanding architectures.

Keyword Codes: D.3.3; F.1; F.2
Keywords: Programming Languages, Language Constructs and Features; Theory of Computation, Computation by Abstract Devices; Analysis of Algorithms and Problem Complexity

1. Theory versus practice?

In much current Computer Science research there seems to be a *quite unnecessary gap* between theory and practice. Work lying on the interface between theory and practice is often subject to skepticism both from practitioner, who may doubt that theory will ever be useful for solving their complex, real-world problems; and from theoreticians, who may be displeased to see their elegant frameworks and concisely stated and powerful results made more complex and less transparent when modified as inevitably is necessary to approach realistic problems.

Some theoretical results seem likely to be forever without practical impact, but others may well eventually have very positive effects. An analogy can be drawn with with context-free grammars, which were long studied as a purely academic discipline, but which eventually led to todays's mature and practically useful technology for parser generation [1].

*This research was partly supported by the Danish Natural Science Research Council (DART project) and an Esprit grant from the European Union (Semantique project, BRA 6809).

2. Frameworks to study program speedup

Of course we all want our programs to run faster. What is a suitable framework to study and solve such problems? This is not easy to answer, as indicated by the following points.

2.1. Interesting but impractical results concerning speedup

The complexity community has produced numerous and interesting results, but it can often be quite hard to distinguish useful results from ones of only academic interest — to separate the wheat from the chaff. One reason for this is an overemphasis on asymptotic complexity, measured exclusively by 'big O' notation and completely ignoring constant factors.

Two theoretical results that can be (and often are) misinterpreted

Proposition *Blum's speedup theorem [2].* Let r be a total recursive function of 2 variables. Then there exists a total recursive function f taking values 0 and 1 with the property that to every program p computing f there exists another program q computing f such that $time_p(n) > r(n, time_q(n))$ for almost all n.

Proposition *The Turing machine constant speedup theorem.* If $\varepsilon > 0$ (no matter how small) and A is decided by a multitape Turing machine `p` in time f, then it is decided by another multitape Turing machine `q` (with a larger tape alphabet) in time $\lambda n\,.\,2n + 18 + \varepsilon \cdot f(n)$.

While Blum's speedup theorem proves that there exist mathematically defined functions with no best program, it does not address the question: *do any natural problems display this behaviour*?

As to the Turing speedup theorem, most complexity literature allows each Turing machine program to have its own alphabet and counts the cost of one instruction as 1 time unit, regardless of the size of the tape alphabet. This unrealistic accounting practice allows proof of the above counter-intuitive theorem, which says that any program running in superlinear time can be made to run faster, in the limit, by any desired constant factor.

Clearly this proof technique is of no practical use, as it amounts to saying that one can double a program's speed by doubling its word size. Even on Turing machines the physical realizability of this trick seems dubious, and it is certainly useless for more familiar machine architectures.

The counter-intuitive nature of the constant speedup theorem motivates one to choose computation models and timings closer to daily experience. A final remark: Turing himself, in his pathbreaking analysis of the nature and limits of computation, actually argued for a *uniform* alphabet size limit.

Asymptotically good but practically questionable algorithms

There is no doubt that the 'big O' notation is useful in understanding growth rates of problem solution time and providing a first classification of algorithms by their speed. On the other hand, asymptotically excellent algorithms may be of small significance unless the constant factors involved in their bookkeeping can be kept down.

A notorious example of this is the rapid progress in the area of 'fast' matrix multiplication, the best time known to this writer being $O(n^{2.38})$. While the exponent is impressive,

the cost of bookkeeping makes these methods practically useless except for unrealistically large matrices.

Similarly, polynomial time algorithms based on Kruskal's theorem and graph minors often have runtimes far in excess of practical usability.

Another area concerns *optimal lambda reductions* with methods by Lamping, Gonthier etc. which are proven optimal by Lévy's definition, meaning that they minimize the total number of parallel beta reduction steps. On the other hand, these methods seem to require bookkeeping work that is quadratic in the number of reductions; so the final answer is not in yet as to whether these methods will aid in implementing functional languages.

2.2. The world of day-to-day programming

In practice, it seems that *programming languages* and what they can (naturally) express are often more important than *machine models*. In principle, of course, one can always reprogram a conceptually clean but inefficient algorithm in, say, C to get maximal exploitation of the computer's abilities. Nonetheless, the choice of programming languages has insidious effects: it determines to a certain extent the way one attacks a problem, as some approaches are much more natural in one language than another, a sort of 'Whorfian hypothesis' in Computer Science. For instance, divide-and-conquer is unnatural in Fortran. Further, some operations are significantly more efficient in one language than in another, if provided at all.

Because of this, it is quite important that we better understand the relations between various programming language constructs and the efficiency of problem solution in them.

Computation time is currently the dominating cost, while memory is secondary. Evidence for this is the success of Prolog and SML/NJ, despite their large memory usage. Further, *near-linear* computation times (as function of input data size) are of paramount importance as data grows in size.

2.3. What is a reasonable model of computation?

The choice of computational abilities is important, e.g. what is the instruction set, can address computation be done or not, can pointers in a cell be changed once the cell has been allocated, are programs self-modifiable, and many others.

Generally speaking the 'unit cost RAM' model is close to daily practice, for instance as in C programming, *provided* memory capacity and word size are not overrun.

What is a realistic time measure?

When discussing polynomial time bounds and the class PTIME, it makes little difference which time measure is chosen. However, these factors become highly relevant if we discuss computability within time bounds small enough to be practically interesting.

In spite of the argument that one should 'charge' time proportional to the address length for access to a memory cell, or a dag or graph node, this is not the way people think or count time when they program. Memories are now quite large and quite cheap per byte, so most programmers need to take little account of the time to access memory in external data storage.

Further, computer memories are carefully designed to make pointer access essentially a constant time operation, so users rarely need to be conscious of address length in order

to make a program run fast enough. In practice computer hardware is fixed: word sizes or memory capacities cannot practically be increased on demand.

A useful analogy is with arithmetic: even though the computer can certainly *not* deal with arbitrary integers, it is carefully designed to model operations on them faithfully as long as they do not exceed, say, 32 bits. Given this fact, programmers have the freedom to assume that the computer faithfully realizes the world of arithmetical calculations, thinking of his or her problem and ignoring the computer's actual architecture unless boundary cases arise.

IOn essence, the logarithmic cost model describes the effect of *increasing data and storage size towards infinity*. This assumption implies a computing model where more and more storage, hardware, or circuitry is needed as data size increases. It thus better models *distributed computing*, e.g. situations involving very large data bases, but not daily practice within a single stored-program computer.

If one wishes to consider expandable hardware, then why not be even more realistic, and account for limitations due to hardware in 3-space, the speed of light, etc.? Interesting, but beyond the scope of this paper.

Traps and possibilities for intellectual cheating

An interesting example: Strassen has proved that the satisfiability problem SAT lies in PTIME, *if* unlimited multiplication allowed at unit time cost and word sizes are unlimited. Similar problems can arise from implicit access to a very large address space,. provided address arithmetic is not limited in some way; and many others are known from the literature.

3. Practically important results from theory

It is certainly true that complexity theory has contributed to our understanding of what *can* and *cannot* be done in practice.

Lower bounds

A particularly dramatic series of theorems establish *lower bounds* for various computational problems. Such a theorem cannot be 'beat': no matter how good one is at programming, data structure design, algorithm analysis, or computability theory, any correct solution must use at least the time or space indicated in the theorem. Unfortunately, few practically interesting problems seem to have hard lower bounds, as most problems with provable lower bounds are constructed by some variation over diagonalization.

Complete problems are studied under the slogan 'if you can't solve problems, at least you can classify them'. It turns out that surprisingly many hard and practically interesting problems are 'sisters under the skin,', so an efficient solution to any one implies an efficient solutions to all problems in the same class.

Best known are the NPTIME-complete problems. A very similar situation exists for PTIME-complete problems — a now standard way to show a given problem to be hard to parallelize is to prove that it is complete for PTIME.

It is interesting that there exist *many natural* NPTIME-complete and PTIME-complete problems. In this complexity domain 'soft' lower bounds, i.e. completeness results rather

than firm lower bounds, are characteristic of more natural problems than 'hard' lower bounds.

General constructions

A very positive aspect of some theoretical constructions is their *level of generality*: instead of solving a single problem, which may or may not be relevant to tomorrow's problems, they set up a broad framework for solving an entire class of problems.

For example, Cook's construction [4] to simulate two-way pushdown automata (2DPDA) on a random access machine is interesting since it shows that one can simulate programs *faster than they run*. This has led to new and nontrivial linear time algorithms (as a function of input size) for a number of problems, for two reasons: first, certain problems are quite naturally solved by 2DPDA, albeit inefficiently; and second, any problem solution by a 2DPDA can be automatically converted into a linear time algorithm by Cook's memoization technique.

Stated in other words, this construction defines an *entire programming language*, all of whose programs can be realized in linear time. Another example with similar overtones but quite different methods is Paige's APTS system [10]. If APTS accepts a problem specification expressed in set-theoretic terms, it will output a program solving the problem in time linear in the sum of the input and output sizes.

Again, this is interesting because mechanical manipulation of specifications can lead to algorithms that are much more efficient than what one would expect from a naive reading of the specification.

4. Some recent practical progress

The *software crisis* is well-known and still painful. Antidotes, none wholly successful as yet, include increased *automated* software production and program optimization. An example is to use of *fewer* and *more general* programs, and to use the computer for automatic *adaptation* of perhaps inefficient but broad-spectrum programs, to construct more efficient ones specialized to a specific range of applications. A key tool for this is partial evaluation.

4.1. Partial evaluation

A partial evaluator is a *program specializer* [8,5]. Given program `p` and partial input `in1`, it constructs a new program $\texttt{p}_{\texttt{in1}}$ which, when run on `p`'s remaining input `in2`, will yield the same result that `p` would have produced given both inputs. It is a special case of program transformation, but emphasizes *full automation* and generation of *program generators* as well as transformation of single programs. The motivation: $\texttt{p}_{\texttt{in1}}$ is often significantly faster than `p`.

The main principle is to gain efficiency by sacrificing generality (presumably, general speedups must involving sacrificing one thing in order to gain something else; Blum's result [2] is almost certainly not a general phenomenon). The field has advanved rapidly in recent years, moving from a 'black art' to an engineering tool.

Partial evaluation also gives a remarkable approach to compilation and compiler generation. Partial evaluation of an interpreter with respect to a source program yields a target program. Thus compilation can be achieved without a compiler, and a target program can be thought of as a specialized interpreter. Moreover, a *self-applicable* partial evaluator can be used both for compiler generation and to generate a a compiler generator.

The range of potential applications is extremely large, and partial evaluation has been used for: pattern recognition, computer graphics by 'ray tracing', neural network training, answering database queries, spreadsheet computations, scientific computing, and for discrete hardware simulation. A typical application is to remove interpretation overhead — and *many* problems have a substantial interpretative overhead.

Huggins and Gurevich's work [7], in this proceedings, uses partial evaluation to derive an evolving algebra from a C program — to extract the program's computational essence, free from implementation details.

4.2. Related program improvement techniques

Partial evaluation is only one among several program transformation techniques. Many are based on automated parts of the Burstall-Darlington methodology, a noteworthy example being Wadler's *deforestation*, to optimize programs by shortcutting the production and consumption of intermediate data structures. This is now being integrated into modern compilers for functional languages, e.g. see the paper by Gill in the current proceedings [15,6]. A related and somewhat earlier techniques is Turchin's *supercompilation* [14,13].

Yet another technique is *memoization*, a special case being Cook's techniques for implementing 2DPDA programs efficiently. This is clearly of potential paractical use.

Space limits unfortunately prohibit a wider overview of this important field of automatic program improvement.

5. Some recent theoretical progress

5.1. A hierarchy within linear time

The *constant speedup theorem*, so well known from Turing machine based complexity theory, has been recently shown false for a natural imperative programming language `I` that manipulates tree-structured data [9]. This relieves a tension between general programming practice, where linear factors are essential, and complexity theory, where linear time changes are traditionally regarded as trivial.

Specifically, there is a constant b such that for any $a > 0$ there is a set X recognizable in time[2] $a \cdot b \cdot n$ but not in time $a \cdot n$. Thus **LIN**, the collection of all sets recognizable in linear time by deterministic `I`-programs, contains an infinite hierarchy ordered by constant coefficients. Constant hierarchies also exist for larger time bounds $T(n)$, provided they are time-constructable.

`I` is a simple Lisp-like imperative language, in which programs have uniformly bounded numbers of fixed number of atoms and program variables. It is strong enough to simulate any multitape Turing machine in real time, and differs mainly in having a fair *cost measure* for basic operations.

[2] where n is the size of the input.

Further, it has been shown that *imperative* programs, *first-order functional* programs, and *higher-order functional* programs all define the same class of linear-time decidable problems.

A result obtained only recently is that *cons-free* programs without time limits can solve exactly those problems in the well-known class LOGSPACE. On the other hand, addition of selective updating (for example `RPLACA/SETCAR` in Lisp) makes a provable difference in problem-solving power, as cons-free programs can decide some problems complete for the provably larger class PSPACE. This example shows that language differences (or architecture differences) can make a difference in problem-solving power.

5.2. Power of various instruction and data models

The paper by Regan in these proceedings [12] investigates the problems that can be solved in linear time, and the question of what is a realistic cost measure.

Paige has shown that surprisingly many problems can be solved in linear time on a pointer machine [10]. On the other hand, Ben-Amram and Galil compare the relative power of pointers versus addresses: in [3] and show that for incompressible data, online simulation of a random access machine by a pointer machine requires a logarithmic amount of extra overhead time.

Most 'sensible' problem representations are all equivalent, up to polynomial-time computable changes of representation. The question of representation becomes trickier when one moves to lower and more practically relevant complexity classes, and especially so for linear time computation. Recent work by Paige on the 'reading problem' [11] shows that data formed from finite sets by forming tuples, sets, relations, and multisets can be put into a canonical and easily manipulable storage form in linear time on a pointer machine. This ensures the independence of many standard combinatorial algorithms from the exact form of problem presentation.

6. Questions worth formulating and trying to solve

The following question types seem particularly deserving of further attention in the context of an invited talk on program speedups in theory and practice.

Do more resources give greater problem-solving ability?

This question has (at least) two natural formulations.

A: can *given* problems be (provably) solved faster by using programs having stronger computational abilities? B: given the same time bound, can *more problems* be solved in that bound, given stronger computational abilities?

Remark: A and B are *not the same*. For instance, A can involve specific problems. A natural example, conjectured to be true: can one prove that a 2DPDA *must* take superlinear time to do string matching? It is known that linear time suffices on a RAM.

Question B can be satisfied by just demonstrating the existence of at least one problem, for example by diagonalization. On the other hand it seems quite hard to find *natural* problems that demonstrate such hard lower bounds.

The relative powers of computer language facilities

These are all questions of type A above. Each will give some insight into fundamental questions regarding the relative efficiency and expressive power of different programming language paradigms.

- Is a pure Lisp-like language intrinsically less efficient than an imperative language such as C?
- Do Prolog's *logical variables* allow faster problem-solving? (A construction that lies somewhere between functional and imperative languages in expressive power.)
- Can some problems be solved faster if pointers can be compared for equality?
- Can some problems be solved faster if programs can contain instructions for selective updating of data structures?
- Does address computation, e.g. indexing and hashing, allow faster problem solving for offline computations? ([3] has shown that it does for online simulation.)

REFERENCES

1. Alfred Aho, John Hopcroft, Jeffrey Ullman: The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.
2. Manuel Blum: A Machine-Independent Theory of the Complexity of Recursive Functions. Journal of the Association for Computing Machinery, vol. 14, no. 2, April 1967, pp. 322-336.
3. Amir Ben-Amram, Zvi Galil: On pointers versus addresses. Journal of the Association for Computing Machinery, vol. 39, no. 3, July 1992, pp. 617-648.
4. Stephen A. Cook: Linear Time Simulation of Deterministic Two-Way Pushdown Automata. Information Processing 71, pp. 75-80, North-Holland Publishing Company, 1972.
5. A. P. Ershov: Mixed Computation: Potential applications and problems for study. Theoretical Computer Science 18, pp. 41-67, 1982.
6. Andy Gill, Deforestation in practice: a theoretically based optimiser for Haskell. *IFIP 94 proceedings.* North-Holland, 1994.
7. Yuri Gurevich and James K. Huggins, Evolving Algebras and Partial Evaluation. *IFIP 94 proceedings.* North-Holland, 1994.
8. Neil D. Jones, Carsten Gomard and Peter Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall International, 425 pp., 1993.
9. Neil D. Jones, Constant time factors do matter. *ACM Symposium on Theory of Computing proceedings.* ACM, pp. 602-611 1994.
10. Robert Paige and F. Henglein: Mechanical translation of set theoretic problem specifications into efficient RAM code - a case study. Lisp and Symbolic Computation, issue 4, nr. 2, pp.— 207-232, August 1987.
11. Robert Paige, Efficient translation of external input in a dynamically typed language. *IFIP 94 proceedings.* North-Holland, 1994.

12. Ken Regan, Linear Speed-Up, Information Vicinity, and Finite-State Machines. *IFIP 94 proceedings.* North-Holland, 1994.
13. Morten Heine Sørensen, Robert Glück and Neil D. Jones, Towards unifying partial evaluation, deforestation, supercompilation, and GPC. *European Symposium on Programming (ESOP).* Lecture Notes in Computer Science, Springer-Verlag, 1994.
14. Valentin F. Turchin, The concept of a supercompiler. *ACM Transactions on Programming Languages and Systems*, 8(3), pp. 292–325, July 1986.
15. Philip L. Wadler, Deforestation: transforming programs to eliminate trees. European Symposium On Programming (ESOP). Lecture Notes in Computer Science 300, pp. 344-358, Nancy, France, Springer-Verlag, 1988.