November 2006 (vol. 7, no. 11), art. no. 0611-oy002
1541-4922 © 2006 IEEE
Published by the IEEE Computer Society

Education

Flexible Algorithms: Overview of a Beginners' Course

R.B. Yehezkael

First courses in computer science usually deal with sequential algorithms and programs. Unfortunately, this gets students accustomed to sequential thinking, making it hard for them to understand and effectively use parallelism later. On the other hand, because parallel programming is so difficult, introducing it at an early stage is impractical. So, my colleagues at the Jerusalem College of Technology and I developed flexible algorithms for teaching.

Flexible algorithms have a simple notation and are multifaceted. You can execute them sequentially or in parallel, producing results independent of the execution order.

Our course aims to teach students to

- read and execute flexible algorithms and understand their behaviors,
- acquire an early awareness of parallelism,
- convert flexible algorithms to hardware block diagrams, *
- convert flexible algorithms to sequential algorithms,
- use computational induction and thereby better understand flexible algorithms,
- make changes to flexible algorithms, and
- write flexible algorithms.

To arouse the students' curiosity and broaden their outlook, we show how flexible algorithms can be converted to hardware block diagrams and briefly show how they can be used for overall system descriptions.

Course development

We described a language, Simple Flexible Language (SFL), that enables flexible execution and built a prototype compiler for it.[1,2] The first version of our course was based on this language but was too complex notationally. So, students and teachers had problems. The second version was notationally simpler and both students and teachers were satisfied. The course notes I describe in this column are based on notes in Hebrew[3] for a Jerusalem College of Technology course, which have been rewritten in English[4] and expanded.

We allocated a total of one lecture hour and one exercise hour per week for the Jerusalem College of Technology course, but the latest expanded course notes might need more time.

Course material

Should the first steps we teach deal with sequential or parallel algorithms and programs? Should they deal with software or hardware? Should we discuss small-scale matters (such as algorithms and logic design) or large-scale ones (such as systems)? Like typical first courses, our approach is small scale, but we differ by starting with flexible algorithms.

Execution methods

We present three execution methods:

- parallel,
- sequential with immediate calls, and
- sequential with delayed calls.

All methods present the execution in the form "sets of statements with values of results." The parallel and sequential execution order is implicit; there are no statements for forcing one kind of execution.

Hardware block diagrams

We present an algorithm for adding serially bit by bit (or digit by digit) and an example of its execution. We then transform this algorithm into a hardware block diagram of the type found in books on logic and digital systems. The techniques for executing and generating a hardware block diagram are similar, and both techniques process sets of statements in similar ways.

Different styles for passing parameters

Here's a fragment of a flexible algorithm written as a function:

```
function v' •= reverse(v, low, high);
{
...
  v' •= reverse(v, low + 1, high -1);
...
} // end reverse
```

This fragment includes a call or activation of itself, which we wrote functionally. We can equivalently write such a call in an abbreviated assignment style, showing only the parameters changed:

```
reverse(low •= low + 1; high •= high -1).
```

The statements `low •= low + 1; high •= high - 1` don't change the values of `low` and `high`. Here, "low" on the right side of the assignment denotes the current variable `low`, and "low" on the left side denotes the new variable `low` that will be used by the new activation of the function `reverse`. The same is true for the variable `high`. So, each call or activation of the function `reverse` has separate variables.

The students quickly caught on to how several variables can have the same name. I explained that it's no more difficult than dealing with a number such as 999, where the leftmost 9 denotes the value 900, the middle 9 denotes the value 90, and the rightmost 9 denotes the value nine. That is, the value of the digit 9 depends on its position or context. Similarly a variable on the right of •= is an existing variable, and one on the left of •= is new.

Conversion to a sequential algorithm

As I mentioned earlier, we illustrate how to convert a flexible algorithm to a sequential one to keep our course from becoming divorced from sequential programming. Using the abbreviated assignment style is important in making the conversion because it provides statements that look on the surface like assignment statements. They're good candidates for being rewritten as actual assignment statements that update a variable's value.

Computational induction

We found computational induction the easiest proof technique, but it doesn't prove that execution has ended However, you can get some confidence that execution has terminated by executing the algorithm on example data (by hand). Also, in the general case, "proof of termination" is a provably unsolvable problem.

Labeled and unlabeled blocks and local variables

Such blocks are convenient abbreviations for internal functions that let us easily introduce local variables and loops. This makes writing algorithms simpler, but we delay presenting this material until the student has mastered the basic form of flexible algorithms and proofs by computational induction.

We also present a `forall` loop and various equivalent forms, including one that enables (but doesn't force) parallel execution of all iterations. For example, `forall i •= m, n; {statements}` is equivalent to this set of unlabelled blocks:

```
{let i •= m; statements}
{let i •= m + 1; statements}
...
...
{let i •= n – 1; statements}
{let i •= n; statements}
```

Each iteration of the `forall` loop has a separate local variable `i`, and this enables parallel execution of all the iterations.

Other approaches

Other educators have tried to use a nonsequential approach with beginning computer science students. At Imperial College London, beginners learn the functional language Haskell and the logic-programming language Prolog, followed by sequential programming in Java. At the Massachusetts Institute of Technology, students learn the Lisp dialect Scheme in a first course. The Warsaw School of Social Psychology's Marcin Paprzycki suggests teaching sequential programming as a specific case of parallel programming.[5]

Reactions to our approach

Some of our colleagues thought that we should teach such a course after students mastered sequential algorithms and programs. Perhaps they were right concerning the course's first version, but not its second version. Indeed, after the college instituted a common first semester for all engineering departments, it decided that our course wasn't of sufficient interest to all departments and moved it to the second semester after the course on sequential algorithms and programs. Teachers then found the course to be too easy, which confirmed that the course material is suitable for beginners.

Another colleague thought the opposite—that such a course should be taught before the course on sequential algorithms and programs.

As I mentioned earlier, students found the first version difficult but enjoyed the second version. Although one student felt the course was superfluous, most comments were positive—for example:

- "This course is better than the companion course on sequential algorithms and programs because there's the possibility of using parallelism."
- "In time, everyone will know languages such as C++ and Java. The knowledge of this kind of material regarding parallelism is an advantage in finding work."

- “It’s amazing that you can derive a hardware block diagram of a serial adder from a flexible algorithm for addition.”

To help students exploit parallelism effectively, it’s important that they learn about it in a simplified form at the beginning of their studies. After taking our course, students could easily execute flexible algorithms in various orders and make small changes to the algorithms. They had little difficulty converting a flexible algorithm to a sequential one. Our notation’s iterative style and a one-time assignment statement made it easier for students to make the transition to iterative or procedural programming. Writing flexible algorithms from scratch was harder for them, as was computational induction.

The course is designed to arouse the student’s curiosity and broaden his or her outlook, which is important for a first computer science course.

References

1. R.B. Yehezkael, “Simple Flexible Language—SFL,” presented at the Compiler Group Meeting, IBM Research Laboratories, May 2000; http://cc.jct.ac.il/~rafi/sfl.pdf (pdf).
2. I. Dayan, “A Prototype Compiler for Simple Flexible Language (SFL),” Jerusalem College of Technology, 2002 (available from R.B. Yehezkael).
3. E. Dashtt, E. Gensburger, and R.B. Yehezkael, “Introduction to Algorithms and Systems,” lecture notes, Computer Science Dept., Jerusalem College of Technology, 2005 (in Hebrew); http://cc.jct.ac.il/~rafi/algosys.pdf (pdf).
4. R.B. Yehezkael, “Flexible Algorithms—An Introduction,” 2006; http://cc.jct.ac.il/~rafi or www.rby.name.
5. M. Paprzycki, “Education: Integrating Parallel and Distributed Computing in Computer Science Curricula,” *IEEE Distributed Systems Online*, vol. 7, no. 2, 2006; http://csdl2.computer.org/comp/mags/ds/2006/02/o2006.pdf (pdf).

R.B. Yehezkael (formerly Haskell) is an independent academic who retired from the Jerusalem College of Technology. Contact him at rafi@jct.ac.il; http://cc.jct.ac.il/~rafi.

Related Links

- DS Online's Web Systems Education Page
- "An Undergraduate Success Story: A Computer Science and Electrical Engineering Integrative Experience," IEEE Pervasive Computing
- "From Research to Classroom: A Course in Pervasive Computing," IEEE Pervasive Computing

* The course's intent was to expose students to examples of converting flexible algorithms to hardware block diagrams, not to teach students how to do such a conversion independently.

Cite this article:
R.B. Yehezkael, "Flexible Algorithms: Overview of a Beginners’ Course," *IEEE Distributed Systems Online*, vol. 7, no. 11, 2006, art. no. 0611-oy002.