Twelve Ways to Ensure that Your Parallel Programming System Doesn't Make Others Look Bad

Gregory V. Wilson (Vrije Universiteit, Amsterdam)
Brent Gorda (Lawrence Livermore National Laboratory)
Paul Lu (University of Toronto)

June 16, 1994

Anyone who has ever programmed a highly parallel computer knows that parallel programming is more difficult than sequential programming. In order to keep ourselves employed, we should make sure it stays that way. Therefore, if you are thinking of adding to the hundreds of parallel programming systems already in existence, please follow the rules given below to ensure that the software you produce does not make the rest of us look stupid, or put us out of a job.

1. **Do not allow users to think about the logical and physical structure of their programs separately.** For example, you should:
   - Require users to do index calculations for distributed arrays by hand; it will encourage efficiency.
   - Require that array dimensions be some multiple of the number of processors in the machine, or a power of two.
   - Discourage users from putting more than one process on each processor by providing a message-passing system in which addresses are (processor, local-process-id) pairs.
   - Use point-to-point channels as the basis for your programming model. Most users enjoy drawing plumbing diagrams for their programs (it reminds them of the good ol' days of unstructured flow charts), and are stimulated by the intellectual challenge of determining whether process (i, j) is supposed to use channel $(i + j)$, and pass channel $(i - j + 1)$ to its first-but-one child, or *vice versa*. Point-to-point channels also complicate the implementation of group operations (discussed below). If you are really keen, you can build a graphical user interface to help users draw plumbing diagrams for their programs. However, you should make sure that you limit its usefulness by (a) not checking the consistency of the plumbing created, or (b) require the user to include physical layout information right from the beginning.
2. **Do not include group operations**, such as barrier synchronization and broadcast. If you do, you will ruin the academic career of at least one graduate student at every user site, who would otherwise have been able to write an entire thesis on how s/he implemented these operations using logical trees. If you cannot

avoid including group operations, try to follow the first rule in this list by making sure that they require the participation of all of the processors in the machine.

3. **Do not support I/O.** Well, `printf` to `stdout` is probably OK, but whatever you do, don't give every process direct access to the file system. I/O just slows programs down, and let's face it: the more test files a program is run on, the more likely it is to turn up a bug in your run-time system. If you do provide I/O facilities, make it the user's responsibility to ensure that input and output are interleaved sensibly. Real users enjoy looking at the detailed workings of systems buffers.

4. **Do not marshal message data automatically.** Packing and unpacking non-contiguous array elements or heterogeneous data into doesn't really have anything to do with parallelism, so why waste your time implementing it? It is much better to require users to pack data into messages manually, since this will encourage them to use only small, simple data structures. If you have the nerve, you can even make this sound like an advantage ("Unlike System Splodge, our system does not incur the overhead of automatic data marshalling...").

5. **Do not support interactive use or timesharing, particularly not on a full-sized machine.** Remember the thrill of sending a box of punched cards down to the Burroughs in the basement for the first time? You can recapture some of that feeling for your users by putting a 1960s-style job queue between them and the machine. Requiring them to wait for their job to come up before they can start debugging re-creates that "old time" atmosphere even more. The lack of time-sharing, and the difficulty of getting the full machine when space-sharing, will also ensure that the only applications which are ever run will be those which were easy to parallelize. Your only users will therefore be ones who think your programming system is great, which is obviously better than having to deal with a mob of ungrateful whiners.

6. **Make sure your system's behavior is as variable as possible.** After all, if a program generates the same result twice, then obviously at least one of those runs has been wasted. One proven technique is not guaranteeing the order of floating-point operations. Another is requiring processes to handle messages in the order of their arrival, rather than allowing them to select messages according to their own criteria. Debugging non-deterministic programs is a rite of passage akin to toggling bootstrap code into a DEC-10, and just as important a part of a programmer's education.

7. **Make it difficult for users to determine what is going on inside their programs.** You would be embarrassed if users ever found out that 72% of their total execution time was spent loading the same executable binary N times, so make damn sure they can't find out. Corollaries of this rule include:

   - Vendors shouldn't include hard-wired LEDs on their machines, since they might show everyone what isn't happening inside. Programmable LEDs are better, since they can be set to flash on and off a lot to simulate a much more desirable "reality".
   - Do not build a debugger for your system. This will not only encourage programmers to write bug-free code, but will also keep the total number

of users down, so that you won't have to spend a lot of time doing support work. Note that it is OK to provide `dbx`-on-a-node, since that won't actually help anyone. It is also OK to provide trace visualization tools, since these are much better at displaying data than at giving programmers useful information.

Remember, all users really want for their papers are speedup figures; absolute performance is unimportant, particularly if you keep repeating:

8. **"A more intelligent compiler could easily optimize this case."** This is a very useful phrase, and you should use it whenever you are discussing your system's (lack of) performance. It is naïve to think that you should include some discussion of whether anyone actually knows how to implement such optimizations, or how effective they would be. Your users will trust you.

9. **Shared object or address spaces are a bad idea.** In fact, shared data structures of any kind are a bad idea. One particularly bad thing about them is their usefulness: users will exercise them in so many different ways that your system is bound to break. Tell them what real FORTRAN (and OCCAM) programmers told advocates of recursion: it's unsafe, and anyway you can emulate it by hand if you really want it.

10. **Keep adding features to your system until any single user can only ever understand a small sub-set of it.** If properly handled, each new feature can be good for at least two conference papers, or another page in the standard with your name on it. Building your system this way will also give implementors lots of places to lay the blame for poor performance. And keep pushing the entire set of features, rather than making simplifications for new users. After all, the fewer people who know how to do this stuff, the more secure our jobs will be.

11. Natural languages, such as Journal English, are ambiguous for a reason. **Be as fuzzy as possible** when describing the status of your system in order to give the impression that something you've been thinking about has in fact been implemented in some version of your system other than whichever one you are presently discussing. Vendors do this sort of thing all the time; why should they have all the fun?

12. Finally, **do not write any non-trivial applications** on top of your system, and try to discourage users from doing it either. Deep down, you intuitively know that your system is a good one; time spent double-checking that intuition is just time taken away from writing the next version. Anyway, the Mandelbrot set (a heavy user of CPU cycles in industry), the Game of Life, matrix multiplication (or LU decomposition without pivoting if you really must show off), and the N-queens problem were good enough for your supervisor; they should be good enough for you.