

Automatic Parallelization is Key for Manycore Success

Position Statement: David August (august@cs.princeton.edu)

Companies make money by alleviating customer pain, not by creating it because they can't solve their own problems. Now that architects are unable to sustain the exponential increase in performance with hardware alone, the next step should be to explore compiler and runtime solutions jointly with hardware, not to simply push the problem to the language level and unduly burden the programmer. The motivation behind this language-level reaction is easy to understand, but it is not well-founded. Worse, it may produce catastrophic results. Some would argue that x86 prevails solely for the reason that switching to another ISA would involve some customer effort. Yet, in the proposed new-language migration path to manycore, we see little respect for this key market insight. The market always has the last word.

New Languages for Manycore Aren't Desirable

In order for any language to be successful, it must reduce the programmer's burden. Unfortunately, new languages for manycore are likely to increase software development burden in at least a few ways. First, a new language of any type requires training, a time consuming and expensive process. Second, legacy codes that are to benefit will need to be rewritten, also a time consuming and expensive process. Third, manycore languages will likely require programmers to express their code in a parallel form, a fundamentally more difficult task than expressing code in a sequential form. Fourth, parallel forms may be more difficult to reason about, resulting in more programming errors. Fifth, compounding this correctness problem will be the increased effort necessary for debugging and verifying parallel codes.

Well-designed manycore languages can address some of these issues by being deterministic and by allowing the decomposition of the problem in a natural manner for the domain. Unfortunately, by supporting a more natural decomposition of the problem, these languages will be special purpose (such as for creating hardware/simulators or for encoding signal processing applications) as different problems decompose in very different ways. When compared to general purpose languages, these languages tend to have higher tool, support, training, and legacy code conversion costs, placing a different set of the burdens on software developers.

New Languages for Manycore Can't Solve the Problem Alone

Even assuming programmers are willing to accept the burden thrust upon them, there is evidence to suggest that new manycore languages won't actually solve the problem without help from automatic parallelization.

When writing code in a manycore language, some algorithms will have programs for which performance scales with input set size. This is good news for manycore if we expect a doubling of cores every Moore's Law generation. These "embarrassingly parallel" applications are easily expressed with existing methods. Unfortunately, many algorithms may not have programmatic expressions which scale so well. Amdahl's law says that the performance of these codes will depend on their "sequential portions".

A "sequential portion" is really just another name for code sections not expressible in a parallel fashion by the programmer. Note that this does not mean that these codes cannot be parallelized, just that they cannot be readily expressed in a parallel manner. Advances in languages may reduce the amount of "sequential portions" in codes, but almost surely at the cost of programmer effort. Consider the disaster that would have resulted if programmers were required to play a part in ILP extraction (dealing with target-specific details, with the fine-grained management of code, and with the distraction from the task at hand). This is what parallelizing "sequential portions" may demand of the programmer (dealing with specific partitionings and mappings to cores, with orchestrating speculation, and with managing more parallelism than is cognitively natural). We have found that the parallelization of "sequential portions" fits the ILP analogy in another manner; these "sequential portions" are often easily parallelized by existing and recently developed automatic parallelization methods.

Amdahl's law means that the success of manycore languages is dependent upon the success of technology to automatically parallelize sequential codes. One might ask: If new manycore languages aren't sufficient, are they even necessary?

The Power of Automatic Parallelization is Underestimated

Some believe, based on results from public and private limit studies, that legacy sequential codes, such as found in the SPEC CPU Integer benchmark suite, would yield at most a 50% performance improvement by automatic means, even when using speculation aggressively. Our recent research paints a very different picture. Consider a sampling of our results on the following page for four sequential programs. Not only do these and our other results invalidate the limit studies, but they show scalable performance for several generations 32. (In these graphs, different lines represent different input sets, the red dots represent the historic performance growth trend for each Moore's law generation, and the speedup on the Y-axis is a multiplier, not a percentage.) As our compilation and architecture technology improves, we expect the performance in these graphs to improve further.

We are learning quite a bit about this problem from our experience with these codes on our compiler and our architecture model. We understand why the often cited limit studies are not a valuable upper bound, we understand why prior efforts to automatically parallelize these codes have failed, and we see a viable path forward.

We believe that we can express "parallel" algorithms sequentially, avoiding all the problems described above for parallel or new manycore languages. We envision a compilation and runtime system that can create the parallel expression, handle communication, and guarantee correctness automatically and by design (assuming a correct sequential expression). This approach has migration, learning, debugging, understanding, determinism advantages. Debugging this sequential expression of a parallel algorithm is no different than debugging sequential codes - it's deterministic and fully ordered. This method also embraces legacy code and provides a smoother migration path for developers.

By understanding the limits of automatic parallelization, we can understand the minimum additional effort necessary from the programmer. While sequential code defines a single outcome for a given input, automatic parallelization would benefit from an understanding of other alternate correct outcomes for that same input. For example, while any sending order for well-formed, sequentially numbered, TCP packets is correct, current systems are forced to reorder these packets before sending even if they are ready to go early. We have experimented with simple methods to provide this support in legacy languages.

Final Word

Vocal proponents of new manycore languages often cite no alternative as the reason. They argue that the multiprocessor automatic parallelization limitations that exist today despite decades of research indicate that manycore automatic parallelization is not a viable solution today. There are at least two reasons why this is an unreasonable leap in logic. First, multiprocessor automatic parallelization is not manycore automatic parallelization. There are opportunities for manycore to move beyond single-chip multiprocessors, including ideas only applicable to manycore that change the problem. Second, advances in compiler technologies developed in the ILP compilation era change the equation. These ILP optimizations are often more effective at extracting TLP than they ever were in extracting ILP. These techniques include new memory analyses, compiler-controlled speculation, and dynamic compilation, to name a few. While we should keep an eye on the past for ideas, we should also understand that manycore automatic parallelization is a completely new challenge, but one with many new possibilities. Promoting new manycore languages as the only solution now is an admission of defeat for computer architects and compiler writers before they have had a chance to focus on the problem.

