Programming Multicores: Do Applications Programmers Need to Write Explicitly Parallel Programs?

In this panel discussion from the 2009 Workshop on Computer Architecture Research Directions, David August and Keshav Pingali debate whether explicitly parallel programming is a necessary evil for applications programmers, assess the current state of parallel programming models, and discuss possible routes toward finding the programming model for the multicore era.

Arvind
Massachusetts Institute of Technology

David August
Princeton University

Keshav Pingali
Derek Chiou
University of Texas at Austin

Resit Sendag
University of Rhode Island

Joshua J. Yi
University of Texas School of Law

Moderator's introduction: Arvind

Do applications programmers need to write explicitly parallel programs? Most people believe that the current method of parallel programming is impeding the exploitation of multicores. In other words, the number of cores in a microprocessor is likely to track Moore's law in the near future, but the programming of multicores might remain the biggest obstacle in the forward march of performance.

Let's assume that this premise is true. Now, the real question becomes: how should applications programmers exploit the potential of multicores? There have been two main ideas in exploiting parallelism: implicitly and explicitly.

Implicit parallelism

The concept of the implicit exploitation of the parallelism in a program has its roots in the 1970s and 1980s, when two main approaches were developed.

The first approach required that the compilers do all the work in finding the parallelism. This was often referred to as the "dusty decks" problem—that is, how to exploit parallelism in existing programs. This approach taught us a lot about compiling. But most importantly, it taught us how to write a program in the first place, so that the compiler had a chance of finding the parallelism.

The second approach, to which I also contributed, was to write programs in a manner such that the inherent (or obvious) parallelism in the algorithm is not obscured in the program. I explored declarative languages for this purpose. This line of research also taught us a lot. It showed us that we can express all kinds of parallelism in a program, but even after all the parallelism has been exposed, it is fairly difficult to efficiently map the

0272-1732/10/$26.00 © 2010 IEEE Published by the IEEE Computer Society

exposed parallelism on a given hardware substrate. So, we are faced with two problems:

- How do we expose the parallelism in a program?
- How do we package the parallelism for a particular machine?

Explicit parallelism

The other approach is to program machines explicitly to exploit parallelism. This means that the programmer should be made aware of all the machine resources: the type of interconnection, the number and configuration of caches in the memory hierarchy, and so on. But, it is obvious that if too much information were disclosed about the machine, programming difficulty would increase rapidly. Perhaps a more systematic manner of exposing machine details could alleviate this problem.

Difficulty is always at the forefront of any discussion of explicit parallel programming. In the earliest days of the message passing interface (MPI), experts were concerned that people would not be able to write parallel programs at all because humans' sequential manner of thinking would make writing these programs difficult. In addition, the more resources a machine exposes, the less abstract the programming becomes. Hence, it is not surprising that such programs' portability becomes a difficult issue. How will the program be transferred to the next generation of a given machine from the same vendor or to an entirely different machine? Today, the issue of portability is so important that giving it up is not really an option. To do so would require that every program be rewritten for each machine configuration. The last problem, which is equally important as the first two, is that of composability. After all, the main purpose of parallel programming is to enhance performance. Composing parallel programs in a manner that is informative about the performance of the composed program remains one of the most daunting challenges facing us.

Implicit versus explicit debate

This minipanel addresses the implicit versus explicit debate. How much about the machine should be exposed? Professors David August and Keshav Pingali will debate the following questions:

- How should applications programmers exploit the potential of multicores?
- Is explicitly parallel programming inherently more difficult than implicitly parallel programming?
- Can we design languages, compilers, and runtime systems so that applications programmers can get away with writing only implicit parallel programs, without sacrificing performance?
- Does anyone other than a few compiler writers and systems programmers need explicitly parallel programming?
- Do programmers need to be able to express nondeterminism (for example, selection from a set) to exploit parallelism in an application?
- Is a speculative execution model essential for exploiting parallelism?

The case for an implicitly parallel programming model and dynamic parallelization: David August

Let's address Arvind's questions directly, starting with the first.

How should applications programmers exploit the potential of multicores? We appear to have two options. The first is explicitly parallel programming (parallel programming for short); the second is parallelizing compilers. In one approach, humans perform all of the parallelism extraction. In the other, tools, such as compilers, do the extraction. Historically, both approaches have been dismal failures. We need a new approach, called the *implicitly parallel programming model and dynamic parallelization*. This is a hybrid approach that is fundamentally different from simply combining explicitly parallel programming with parallelizing compilers.

To understand this new approach, we must understand the difference between "explicit" and "implicit."

Consider the __INLINE__ directive. You can, using some compilers, mark a function with this directive, and the compiler will automatically and reliably inline it for you. This directive is explicit. The tool makes

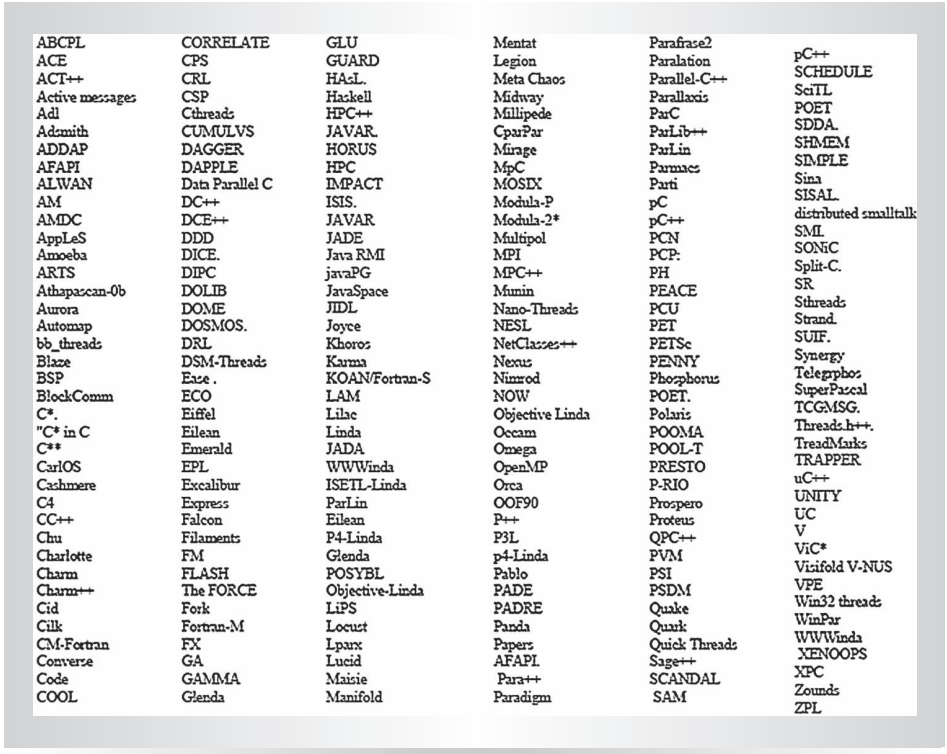

Figure 1. A partial list of parallel programming languages.

inlining more convenient than inlining manually. However, it is not as convenient as simply not having to concern yourself with questions of inlining, because you are still responsible for making the decision and informing the inlining tool of your decision. Explicit inlining is now unnecessary because compilers are better at making inlining decisions than programmers. Instead, by using functions and methods, programmers provide compilers with enough information to decide on their own what and where to inline. Inlining has become implicit since each function and method by itself (without the __INLINE__ directive) is a suggestion to the compiler that it should make a decision about whether to inline.

Parallel programming is explicit in that the programmer concretely specifies how to partition the program. In parallel programming, we have many choices. Figure 1 shows a partial list of more than 150 parallel programming languages that programmers can choose from if they want to be explicit about how to parallelize a program. Somehow, despite all these options, the problem still isn't solved. Programmers have a hard time making good decisions about how to parallelize codes, so forcing them to be explicit about it to address the multicore problem isn't a solution. As we'll see, it actually makes the problem worse in the long run. That's why I'm not a proponent of explicitly parallel programming.

Is explicitly parallel programming inherently more difficult than implicitly parallel programming? For this question, I'm not going to make a strong argument. Instead, my opponent (Keshav Pingali) will make a strong argument for me. In his PLDI (Programming Language Design and Implementation) 2007 paper,[1] he quotes Tim Sweeney, who "designed the first multithreaded Unreal 3 game engine." Sweeney estimates that "writing multithreaded code tripled software code cost at Epic games." This means that explicitly parallel programming is going to hurt. And, it's going to hurt more as the number of cores increases. Through Sweeney, I think my opponent has typified the universal failure of explicitly parallel programming to solve the problem.

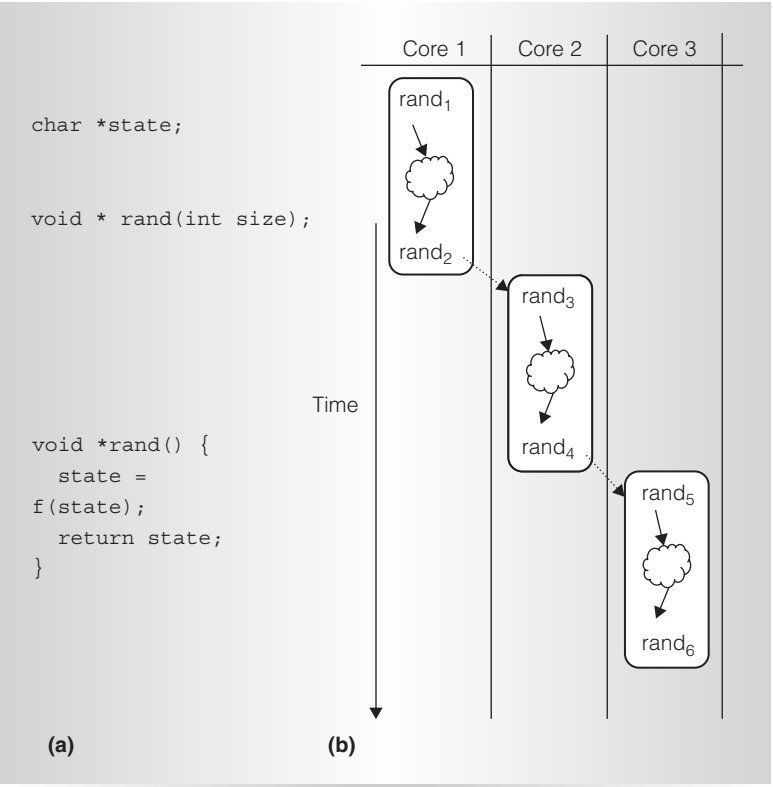

Figure 2. Parallelization diagram: pseudocode (a) and dynamic execution schedule (b). The dependence between each `rand` call creates a serialization of the calls to `rand`.

To support the case for implicitly parallel programming languages, I refer to the same paper by my opponent, in which he says "it is likely that more SQL programs are executed in parallel than programs in any other programming language. However, most SQL programmers do not write explicitly parallel code . . . ." This is a great example of implicitly parallel programming offering success in the real world. What we should be doing as a community is broadening the scope of implicitly parallel programming beyond the database domain.

Can we design languages, compilers, and runtime systems so that applications programmers can get away with writing only implicitly parallel programs, without sacrificing performance? I'd like to answer this question by describing an experiment we've done. In Figure 2, we have a function called `rand`. It's just a pseudo number generator with some internal state, called `state`. Elsewhere, we have an apparently DoAll loop that calls `rand`. Unfortunately, a dependence existing between each `rand` call must be respected, creating a serialization of the calls to `rand`. Scheduling iteration 1 on core 1, iteration 2 on core 2, and iteration 3 on core 3 provides no benefit as `rand` is called at the start and end of each iteration. This serializes each iteration of the loop with the next.

With this constraint, parallelism seems impossible to extract, because this dependence must be respected for correct execution, yet we can't speculate it, schedule it, or otherwise deal with it in the compiler. As programmers, though, we know that the `rand` function meets its contractual obligations regardless of the order in which it's called. The value order returned by `rand` is irrelevant in our example. The sequential programming language constrains the compiler to maintaining the single original sequential order of dependences between calls to `rand`. This is unnecessary, so we should relax this constraint.

To let the programmer relax this constraint, we add an annotation to the sequential programming model that tells the compiler that this function can be invoked in any order. We call this annotation `commutative`, referring to the programmer's intuitive notion of a commutative mathematical operator. The `commutative` annotation says that any invocation order is legal so long as there is an order. Armed with this information, the compiler can now execute iterations in parallel, as Figure 3 shows. The actual dependence pattern becomes a dynamic dependence pattern. The order in which `rand` is called is opportunistic, and large amounts of parallelism are exposed. The details of the `commutative` annotation are available elsewhere.[2]

There are several important things to note. First, the annotation is not an explicitly parallel annotation like an OpenMP pragma. Second, the annotation is not a hint to the compiler to do something that it could possibly figure out on its own if it were smart enough, like the register and inline keywords in C are now. It is an annotation that communicates information about dependences to the compiler, information that is only

knowable through an understanding of the real-world task. Additionally, although we arrived at this use of `commutative` by examining this program's execution plan, we don't expect the programmer to have to do so. Instead, we expect the programmer to integrate `commutative` directly into the programming process, where it should be annotated on any function that is semantically commutative. That is, `commutative` is a property of the function as used in this program, not a property of the underlying hardware, the execution plan, nor any of the other concerns explicitly parallel programmers must consider.

The idea isn't new; my opponent also refers to this idea in his PLDI 2007 paper and talks about "some serial order" and "commutativity in the semantic sense." Note that our `commutative` isn't inherently commutative (like addition), but semantically commutative. So, it states that receiving random numbers out of order, as Figure 3 shows, is acceptable for this program's task even though it might not be acceptable for other tasks. For our implicitly parallel programming experiment, we took all of the C programs in the SPEC CPU 2000 integer benchmark suite and modified 50 of the roughly 500,000 lines of code (LOC). Most of the modifications involved adding the `commutative` annotation. This annotation adds a little nondeterminism in the form of new legal orderings. As Figure 4 illustrates, by using this method, we restored the performance trend lost in 2004.

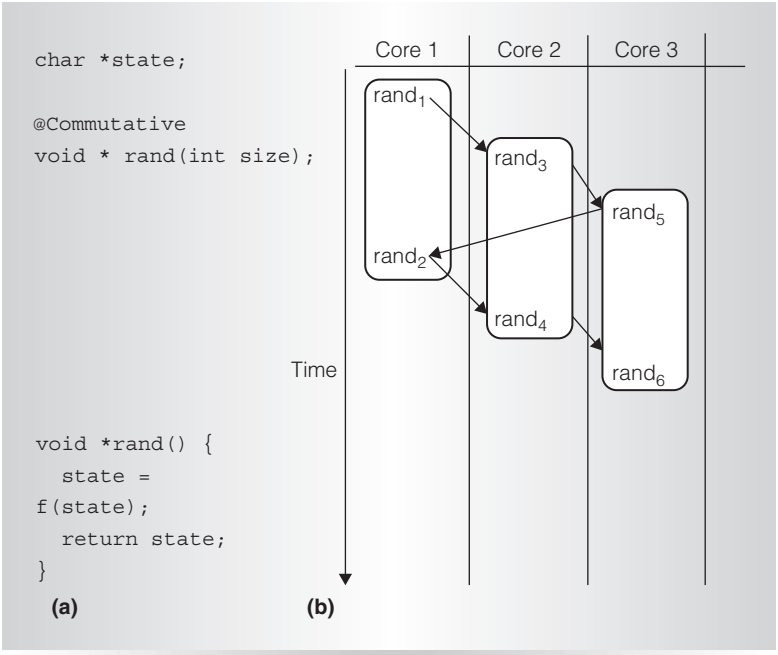

Figure 3. Parallelization diagram: pseudocode with `commutative` annotation (a) and dynamic execution schedule (b). Since the value order returned by `rand` is irrelevant in the example given in Figure 2, we can relax the constraint on maintaining a single original sequential order of dependences between calls to `rand`. We do this by adding the `commutative` annotation to the sequential programming model that tells the compiler that this function can be invoked in any order. With this annotation, the compiler can now execute iterations in parallel.

Do programmers need to be able to express nondeterminism (for example, selection from a set) to exploit parallelism in an application? The answer is yes. Figure 5 shows performance with and without those annotations— the effect of those 50 lines of code out of 500,000. Without the annotations, we can only talk about speedup in percentages, such as 10 or 50 percent—what one normally expects from a compiler. Using the annotations with today's compiler technology, we can talk about speedup in terms of multiples, such as 5 or 20 times.

Is a speculative execution model essential for exploiting parallelism? Yes, the compiler is speculating a lot, but we don't expose it to the programmer. Speculation management can and should be hidden from the programmer, just as speculative hardware has done for the last few decades.

And, really the final question is: Does anyone other than a few compiler writers and systems programmers need explicitly parallel programming? Obviously, compiler writers and system programmers have to do it. For now, they will have to deal with these issues. For everyone lucky enough not to be compiler writers or systems programmers, my answer is: No, you don't need explicitly parallel programming.

My opponent might argue, as he does in his paper, that the programmer must provide the means to support speculation rollback. Well, in our approach, we didn't need that. He might also argue, as he does in his paper, that a need for opportunistic

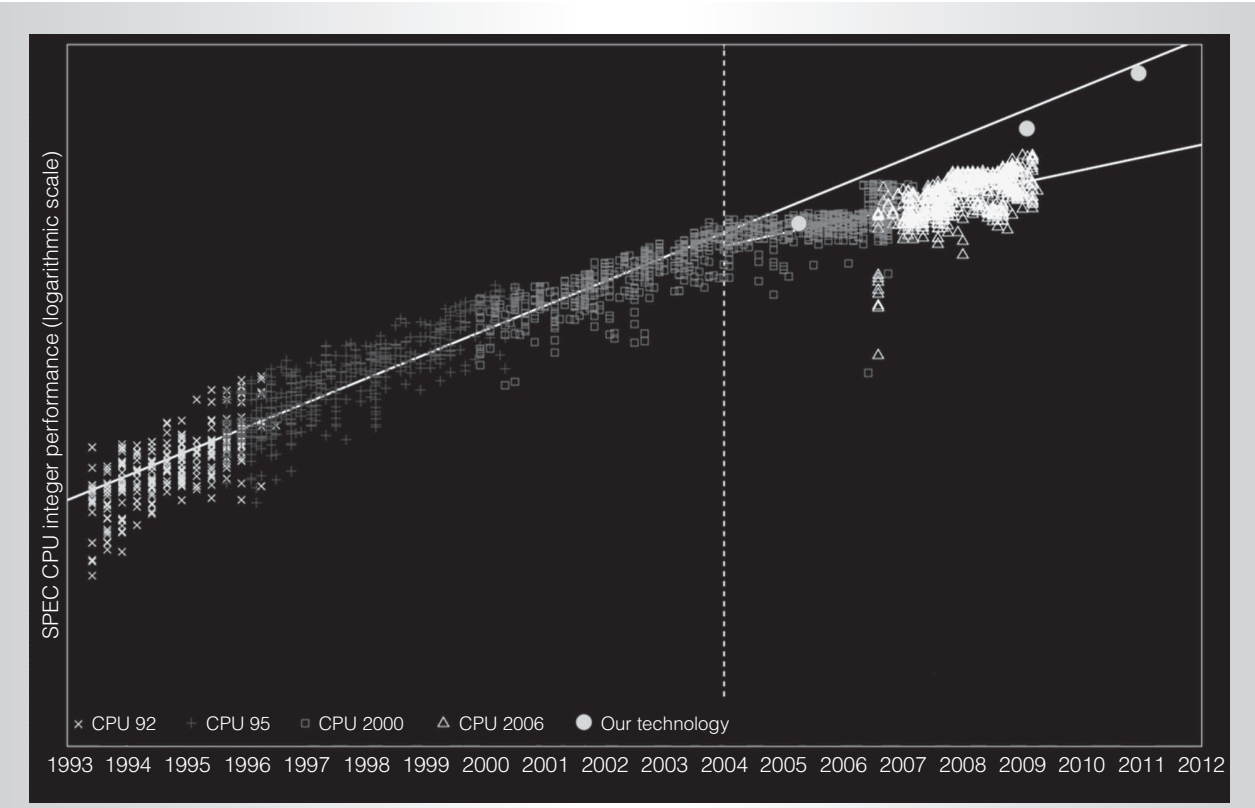

Figure 4. Restoration of performance trend lost in 2004. The compiler technology in this figure is the implicitly parallel approach developed by the Liberty Research Group, led by David August at Princeton University.

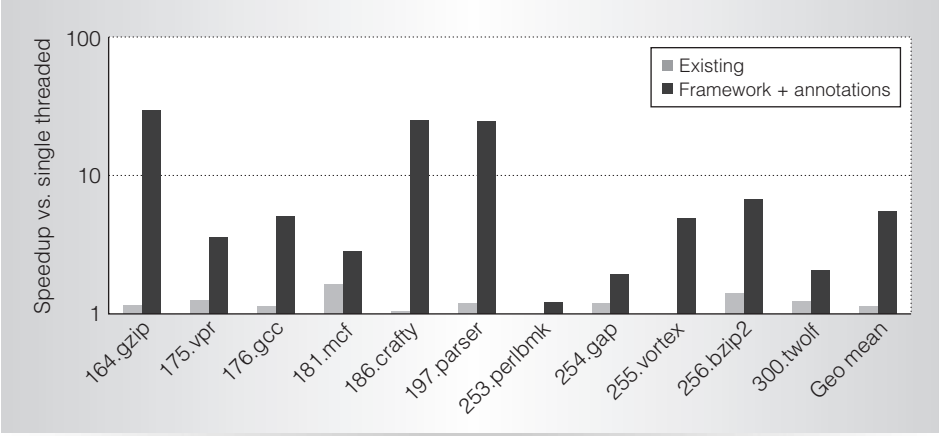

Figure 5. Performance with and without the implicitly parallel approach with annotations. Approximately 50 of 500,000 lines of code have been modified.

parallelism exists. As our results in Table 1 show, our implicitly parallel approach provides less than 2 times speedup on some programs. You might think that this is a failure of implicitly parallel programming and that falling back to explicitly parallel programming is the only way to speed up perl. That would be a disaster. Given more time,

I could tell you how we can speed that up without falling back to explicitly parallel programming. This leads me to one final thought.

I have an ally in this debate in the form of the full employment theorem for compiler writers. It says, "Compilers and runtime systems can always be made better." Every time you see a tool fail to make good use of implicit parallelism, you can fix it. You can always make it good enough to avoid reverting back to explicit parallelism and all of its problems. The human effort that went into improving a single application's performance could have been better applied to improving the tools, likely improving the performance of many programs, existing and not yet written.

The case for an explicitly parallel programming: Keshav Pingali

I believe that applications programmers do need to write explicitly parallel programs. I can summarize my thoughts in three points.

First, explicitly parallel programming is indeed more complex than implicitly parallel programming. I think we all agree about that. Unfortunately, I think it is a necessary evil for applications programmers, particularly for those who write what we call "irregular" programs (mainly programs that deal with very large pointer-based data structures, such as trees and graphs).

Second, exploiting nondeterminism is important for performance, and by that I don't mean race conditions. What I mean here is what experts call "don't-care nondeterminism." There are many programs that are allowed to produce multiple outputs, and any one of those outputs is acceptable. You want the system to figure out which output is the most efficient to produce. This kind of nondeterminism is extremely important. I think David pretty much agrees with that, so I won't spend too much time on it.

Finally, optimistic or speculative parallel execution is absolutely essential. This is a point that many people don't appreciate, even those in the parallel programming community. Parallelism in many programs depends on runtime values. And therefore, unless something is done at runtime, such as speculation, we really can't exploit parallelism in those kinds of programs.

Table 1. Maximum speedup achieved on up to 32 threads over single-threaded execution and minimum number of threads at which the maximum speedup occurred.

Benchmark	No. of threads	Speedup
164.gzip	32	29.91
175.vpr	15	3.59
176.gcc	16	5.06
181.mcf	32	2.84
186.crafty	32	25.18
197.parser	32	24.5
253.perlbmk	5	1.21
254.gap	10	1.94
255.vortex	32	4.92
256.bzip2	12	6.72
300.twolf	8	2.06
GeoMean	17	5.54
ArithMean	20	9.81

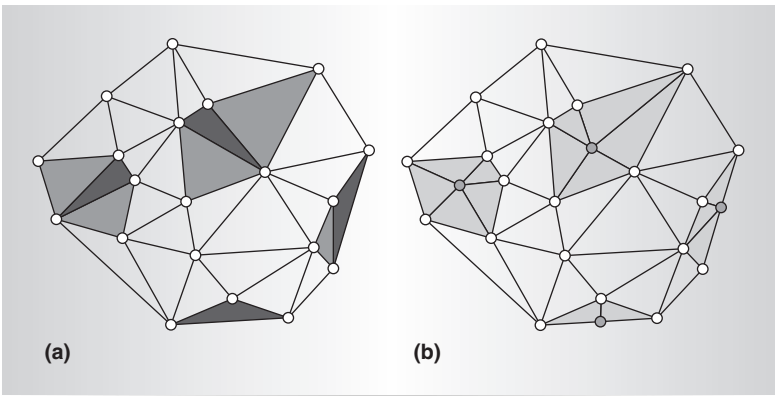

Figure 6. Delaunay mesh refinement example. Through an iterative refinement procedure, triangles within the neighborhood, or cavity, of a badly shaped triangle (dark gray triangles in (a)) are retriangulated, creating new triangles (light gray triangles in (b))

Delaunay mesh refinement

I find it easiest to make arguments with concrete examples. A popular example in the community is Delaunay mesh refinement (DMR).[3] The input to this algorithm is a triangulation of a region in the plane such as the mesh in Figure 6a. Some of the triangles in the mesh might be badly shaped according to certain shape criteria (the dark gray triangles in Figure 6a). If so, we use an iterative

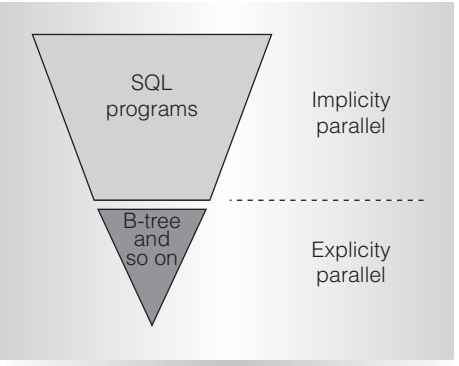

Figure 7. A framework for understanding the implicitly and explicitly parallel programming models. Applications programmers write SQL programs, which are implicitly parallel, and they rely on implementations of relations, such as B-trees, which systems programmers have carefully coded in parallel.

refinement procedure to eliminate them from the mesh. In each step, the refinement procedure

- picks a bad triangle from the worklist,
- collects several triangles in the neighborhood (the *cavity*) of that bad triangle (dark gray regions in Figure 6a), and
- retriangulates the cavity, creating the light gray triangles in Figure 6b. If this retriangulation creates new badly shaped triangles, they are added to the worklist.

There is room for nondeterminism in this algorithm because the final mesh's shape depends on the order in which we process the bad triangles. But we can show that every processing order terminates and produces a mesh without badly shaped triangles. The fact that any one of these output meshes can be produced is important for performance.

Another important point is that parallelism in this application depends on runtime values. Whether bad triangles can be refined concurrently depends on whether their cavities overlap. Because dependences between computations on different bad triangles depend on the input mesh and on the modifications made to the mesh at runtime, the parallelism can't be exposed by compile-time program analyses such as points-to or shape analysis.[4,5] To exploit this "amorphous data-parallelism," as we call it, it's necessary in general to use *speculative* or *optimistic* parallel execution.[1] Different threads process worklist items concurrently, but to ensure that the sequential program's semantics are respected, the runtime system detects dependence violations between concurrent computations and rolls back conflicting computations as needed. (DMR can be executed in parallel without speculation by repeatedly building interference graphs and finding maximal independent sets, and so on, but this doesn't work in general.)

Framework

David and I disagree about what to call these kinds of applications. The framework in Figure 7 is useful for thinking about the implicitly and explicitly parallel programming models.

Let's go back to David's SQL programming example. Database programming, in my opinion, is a great example of an area where implicitly parallel programming succeeded. But it's important to remember that there are two kinds of programs and two kinds of programmers. Applications programmers write SQL programs that are implicitly parallel. So, I accept that this SQL programming has been a success story for implicitly parallel programming. However, application programmers rely on implementations of relations, such as B-trees, which have been carefully coded in parallel by a small number of systems programmers. The question is whether we can generalize this to general-purpose parallel programs. I think about it in terms of Niklaus Wirth's equation: `program = algorithm + data structure`. We need to think about an application in terms of both the algorithm, which the applications programmer codes using an abstract data type; and the concrete data structure that implements that abstract data type. In database programming, algorithms are indeed being expressed with an implicitly parallel language such as SQL. However, the data structures are being implemented with an explicitly parallel language such as C++ or

Java. Can we make this software development model general purpose? We have had some success doing so; however, we've also run into some stumbling blocks, which makes us more skeptical about this particular model.

A programming model example and the need for explicitly parallel programming

A recent effort on the parallel execution of irregular programs (which are organized around pointer-based data structures such as trees and graphs) has led the development of the Galois programming model.[1,6,7] This programming model is sequential and object-oriented (sequential Java). Applications programmers write algorithms using Galois set iterators, which essentially give them nondeterministic choices, sets, and so on. They then rely on the equivalent of a collections library, in which all the data structures, such as graphs and trees, are implemented. An applications programmer can easily write an algorithm, such as DMR, with this sequential language. However, they're relying on a collections library that someone else implemented using explicitly parallel programming with locks and so on. Is this good enough? In other words, can we ship a collections library of data structures that have been implemented with explicitly parallel programming that applications programmers can then use in their sequential object-oriented models and thus avoid having to write any concurrent programs at all? Unfortunately, here we run into three fundamental problems, which make me skeptical as to whether this programming model will work.

The first problem is that no current fixed set of data structures meets all applications' needs. For example, the Java collections library contains many concurrent data structures, but there are many other data structures that aren't in there, such as kd-trees and suffix trees. As library writers interact with applications programmers, they find more of these data structures. They then have to code concurrent versions using explicitly parallel languages. Just to give you an idea of how many data structures there are, a Google search on ''handbook of data structures'' returns a great book by Sartaj Sahni.[8] It's 1,392 pages and lists hundreds of data structures. What that means is that there's a large variety of data structures that we must code in terms of some explicitly parallel programming language. We might then be able to use the Galois programming model on top to express the algorithm. If the data structures that we need to implement the algorithm aren't supported in the library, however, they must still be coded in explicitly parallel programming languages.

Second, even generic data structures must be tuned for a particular application. For example, if you need a graph, you find it in the library and use it for your program. However, if your application has some properties that you must exploit to get good performance, you have no option but to start mucking around with the data structure implementation, which is of course in an explicitly parallel language.

The third problem is tied to the first two. In many applications, the API functions of the various classes (graph, tree, data structure, and so on) do relatively little computation. In databases, this model works well because most of the execution time is spent in library functions that system programmers have carefully implemented in explicitly parallel languages. Hence, there's no need to worry too much about the top-level program. However, many general-purpose applications don't spend a lot of time on data structure operations. Therefore, to get really good performance, you need to squish the application code and data structure implementation together in some way. Unfortunately, we don't currently have the compiler technology to generate good code of that kind from a separate specification of the algorithm and the data structure.

We run into these three issues when we try to shield applications programmers from the complexities of explicitly parallel programming. Are these fundamental problems? If they are, you have no choice but to write some amount of explicitly parallel code even if you're an applications programmer. I see no solutions for these problems right now. So, I'm putting these issues up for debate and will see if my opponent has a viewpoint on them.

Parallel libraries and automatic parallelization

August: There's a lot of talk about solving the parallel programming model problem through the use of libraries. It occurs to me, especially after hearing that one book lists numerous data structures, that there are more libraries than programs. If that's so, maybe the problem is not how to write parallel programs, but how to write parallel libraries. So, we're back to where we started. We need a solution that works for writing libraries and applications.

Pingali: I don't think that's going to solve anything. I agree that explicitly parallel programming is difficult. Therefore, we want to minimize the amount of code written explicitly parallel. However, if applications programmers want really good performance, they might need to dig for data structure implementations, whether or not thousands of them exist. No technology that I know of lets you write these optimized data structure implementations for particular applications. For example, I don't know of any automatic way of taking some implicitly parallel descriptions of data structures, such as kd-trees and suffix trees, and producing optimized parallel implementations. Currently, the only way is to write explicitly parallel code. Transactional memory might make programming those kinds of data structures easier, but ultimately you don't get the performance you want. And, even for the transactions, there's some notion of explicit parallelism.

August: If we were to explicitly parallelize a large number of libraries, we would start to see some patterns, and if we start to see patterns, effort might be better spent implementing ways to automate those patterns by merging those patterns that are in parallel form in a compiler.

In the past, when compilers were the 15-percent solution, the economics weren't there for many of us to get involved in writing automatic parallelization techniques or transforms. But, now that we have a choice between relying on some kind of system or turning to the recurring expense of writing explicitly parallel programs, the economics might be there to take that approach. What do you think?

Pingali: That's an interesting idea, David. I would be happy about this because I don't particularly like explicitly parallel programming, either. However, almost all of the compiler technology developed over the past 40 to 50 years has been targeted toward dense matrix computation. We know very little about how to analyze irregular pointer-based data structures (such as large trees and graphs). Some work has been done, but it hasn't gone anywhere, in my opinion. So, I've become somewhat skeptical about the possibility of developing compiler technology for generating a parallel implementation for these data structures from high-level specifications. But it's an interesting goal.

August: Perhaps we should give up the idea that we can fully understand code statically, an idea that I think has misdirected effort in compiler development for decades. Let's refer to the poor performer benchmark, perl, in our implicitly parallel program experiment. In perl, the parallelism is fully encoded in the input set. The perl script has parallelism in it, but, most likely, the input defines where the parallelism is. Therefore, no static analysis can tell you what the input to that program is or where the parallelism will be because they haven't yet been determined. Therefore, maybe we should think about this as the fundamental problem and move to a model where we look at the code and do transformations at runtime or generationally. We should abandon doing things like shape analysis and heroic compiler analyses in general.

Where should it be done in the software stack?

Audience: I never thought that parallelism would be a problem, at least for many applications. The problem arises in actually running these programs effectively on a machine. A lot of that has to do with locality. For example, where to get data, where to get the structures, how quickly they're obtained, and how much bandwidth is obtained. Unfortunately, I haven't seen anything in the discussion so far to address that. I was just wondering first of all whether

your systems work. That's what was so good about an API. It's terrible that you have to do everything yourself but it really works when you're sure things are in the right place. What are your thoughts on this?

August: From the implicitly parallel programming model viewpoint, we're going to rely on a dynamic component—a layer between the program and the multicore processor—to do the tuning. There has been some success in the iterative compilation world. Compilers don't know the right thing to do unless they can observe the effects of their transformations. Once they have the chance to do so, they can make increasingly better decisions. The goal there is to reduce the burden on the applications programmers to do that kind of tuning.

Arvind: Just for clarification, are you talking about an interactive system?

August: Because of the problem that I mentioned with the perl benchmark, we need to deal with the program after it starts running. We want to observe dependences on values, to speculate upon them, and to tune. We also want to understand if there are other things happening dynamically in the system, such as might reduce the effective core count. We'd like to adjust for such events as well. So, this is a runtime system.

Arvind: Is this for future program runs?

August: It could be the same run. We have extra cores, so we can expend some effort adjusting the program. Some of the most successful models are those in which the static compiler only partially encodes the program and leaves the rest to be adjusted at runtime. It isn't a dynamic optimizer as such since the program isn't fully defined. Instead, the program can be parameterized by feedback from the performance at runtime. So, it's a hybrid between a compiler running statically and purely runtime optimization.

Pingali: One mistake we keep making in the programming languages and architecture community, or the systems community more broadly, is thinking about programmers as monolithic because we're used to the idea of SPEC benchmarks. So, there is one big piece of code and someone has to put something in that C (or other language) code to exploit parallelism. We need to get away from this kind of thinking because domains in which people have successfully exploited parallelism have two classes of programmers. There are "Joe" programmers—applications programmers with little knowledge about parallel programming although they're domain experts. And, there are "Steve" or "Stephanie" programmers—highly trained parallel programming experts who know a lot about the machine, but little about a particular application domain. These programmers write the middle layers.

In the database domain, for example, Joe programmers write SQL code. They don't know a lot about concurrency control and all that other stuff, but they get the job done. There are a small number of people at IBM, Oracle, and other corporations, who implement, for example, WebSphere carefully and painfully in C++ or Java. Therefore, clearly we need to manage resources and locality to get good performance. The real question we should be asking is where in the software stack this management should take place. If there's any hope for bringing parallelism into the mainstream and for having millions of people write programs, we can't leave resource and locality management to applications programmers. Rather, it must be done at a lower level, even if it's less efficient than doing it with respect to having some knowledge about the application. So, I don't think we have any choice. We shouldn't be considering whether to do it, but rather, where in the software stack to do it and how to make sure that as much as possible is done lower in the stack. Otherwise, multicores will never go mainstream.

August: That sounds like an argument for implicit programming.

Pingali: We should do as much implicit programming as we can at the Joe programmer level. There is no debate about that. The question is whether we can get away from explicitly parallel programming altogether, and my position is no.

Are libraries really the solution?

Audience: Professor Pingali, you were talking about how people always handcraft libraries. Let's talk about the C++ Standard Template Library. In many software companies today, if you handcraft data structures or operations that can't be expressed using STL, you're fired.

Pingali: We still haven't dealt with parallelism in regard to STL. So, can we build a parallel implementation of STL that covers such a broad application scope that companies can really start firing people the moment they say they're getting terrible performance for a particular application and they need to code something at the STL level using explicitly parallel programming? I don't think we're there yet. I don't know if we'll ever get there for reasons I already mentioned.

Audience: The flip side of this is that STL consists not only of data structure manipulations, linked lists, and so on, but also of math applications such as those applying an operation in mathematical parallel. Almost no one uses math operations in the STL library. Professor August, can your compiler convert some of the code used in this area by employing the math implementation?

August: Let's take an example of an iterator traversing a linked list. We wish that a programmer had used the array library. We also wish that the access order didn't matter—random access to the array elements would be nice, especially in the DoAll style of parallelism. But that doesn't stop us from finding parallelism. Perhaps we can't think of a more serial operation than the traversal of a linked list. But, there are techniques that can transform serial data structures, such as a linked list, into other data structures that enable parallelism, such as arrays or skip lists specific to particular traversals. There are some automatic techniques to deal with these kinds of problems.

Paradigm shift and education problem

Audience: To roughly characterize the two arguments, both of you are saying that explicit programming is hard. One of you argues that it's so hard we shouldn't use it. The other argues that it's hard but we have to do it. I'm curious about the quote on the UnReal engine that David mentioned. How difficult was developing or building the parallel version of that engine for the first time? How much of this is really a transient effect? How much extra programmer time and cost did it entail the next time they tried to build an engine like that? Is this really an education problem because it's a big paradigm shift from single-threaded to multithreaded code?

August: Maybe one thing that will happen is that programmers will change. Perhaps there won't be any Joe programmers anymore. That's possible, but I doubt it. The other thing that we know will change is the number of cores. We're talking about 4, 6, and 8 now. What will happen when we have 32, 64, or more? What about thousands? The multithreaded version of the UnReal engine, which was too hard to build the first time, needs to be built again. The first version didn't explore speculation; however, now you have to do speculation because we need much better speedups than only 3 times. It seems like just as you solved the problem, the microprocessor manufacturers made the problem harder. So, we're continually chasing the rabbit, and there must be a better way.

Audience: Well, that implies that eventually you must do explicit programming. Unless you're saying that your implicit infrastructure can do a better job?

August: Yes, that's my claim.

Pingali: The way I see it, the fewer opportunities for making errors and bugs, the better off Joe programmers are. Perhaps with enough training, we can even get them to write reasonably good concurrent code. But the point is, it's almost always better if you don't have to worry about these concurrency bugs, even if you're Steve or Stephanie programmer. So, what we should do is try to move concurrency management somewhere in the software stack, where it can be dealt with in a controlled way as opposed to just letting it out and saying everyone must

become a Stephanie programmer. I don't think that's going to happen because it hasn't happened in a few domains, such as dense numerical linear algebra, that successfully harnessed parallelism. In the dense numerical algebra domain, a small number of people write matrix factorization libraries, and a large number of Matlab programmers who simply write Matlab use these libraries. If you start looking at the library implementations, such as the matrix factorization code and matrix multiplication, the code is somewhat horrific. To have everyone write their own explicitly parallel code isn't a scalable model of software development.

Implicit versus explicit

Audience: David, you have annotations that you need to add to your programs. Where do you draw the line between implicit and explicit?

August: If the annotation brings information that can't be determined otherwise, it's implicitly parallel. If a compiler or runtime system can determine the existence of dependence automatically, annotations confirming its existence or indicating what to do about it are explicit. Consider the situation where you can give the user the numbers from a pseudorandom number generator in any order, changing the program's output. If the programmer states that this is legal, that statement is implicit because it's otherwise indeterminable. The compiler can't know whether the user needs the numbers in the original order, because sometimes the order is important and sometimes it isn't, depending on the real-world task. To me, that isn't only an implicitly parallel annotation, but it's a good language annotation since it isn't automatically determinable.

How should parallel programming be introduced in a computer science and engineering curriculum?

Arvind: When I was an assistant professor, I went to the department head and said, I have to teach this course, it's very important. He asked how important; I said very important. Then, he said, why don't you teach it at the freshmen level. I said no, no, it isn't that important. So, the question really is how important is parallel programming? Is it separate from sequential programming? And within that context, how important is explicitly parallel programming? Should we teach it at the freshmen level, or is it like quantum mechanics, where you must master classical mechanics before you can touch quantum mechanics? I would like both of you to express your opinion on this.

August: It's as important as computer architecture. So, it should be taught at the junior/senior level.

Pingali: I'd like to go back to Niklaus Wirth's equation: `program = algorithm + data structure`. We need to teach algorithms and data structures first. When teaching students sequential algorithms and data structures, we can also teach them parallelism in algorithms, which we can do without talking about a particular programming model or implementation of locks, transactional memory, synchronization, and so on. We can talk about locality, parallelism, and algorithms without talking about all of those other things. After that, we should introduce implicitly parallel programming at the Joe programmer level. That is, Joe programmers write these Joe programs using somebody's data structures in libraries, which are painfully written, like a collection of classes written over many years. This way, they get a feel of what they can get out of parallel programming. Then, you take the covers off and tell them what's in these data structure libraries, and expose them to their implementations. That's the sequence that we have to go through. I would start with data structures and algorithms at the abstract level. Then, show them implicit parallel programming at the Joe programmer level using parallel libraries. Then, expose them to its implementation. MICRO

References

1. M. Kulkarni et al., ''Optimistic Parallelism Requires Abstractions,'' *Proc. ACM SIGPLAN Conf. Programming Language Design*

and Implementation (PLDI 07), ACM Press, 2007, pp. 211-222.
2. M. Bridges et al., ''Revisiting the Sequential Programming Model for Multi-Core,'' *Proc. 40th Ann. IEEE/ACM Int'l Symp. Microarchitecture,* IEEE CS Press, 2007, pp. 69-84.
3. L. Paul Chew, ''Guaranteed-Quality Mesh Generation for Curved Surfaces,'' *Proc. 9th Ann. Symp. Computational Geometry* (SCG 93), ACM Press, 1993, pp. 274-280.
4. L. Hendren and A. Nicolau, ''Parallelizing Programs with Recursive Data Structures,'' *IEEE Trans. Parallel and Distributed Systems,* vol. 1, no. 1, Jan. 1990, pp. 35-47.
5. M. Sagiv, T. Reps, and R. Wilhelm, ''Solving Shape-Analysis Problems in Languages with Destructive Updating,'' *ACM Trans. Programming Languages and Systems,* vol. 20, no. 1, Jan. 1998, pp. 1-50.
6. M. Kulkarni et al., ''Optimistic Parallelism Benefits from Data Partitioning,'' *Proc. Conf. Architectural Support for Programming Languages and Operating Systems* (ASPLOS 08), ACM Press, 2008, pp. 233-243.
7. M. Kulkarni et al., ''Scheduling Strategies for Optimistic Parallel Execution of Irregular Programs,'' *Proc. Symp. Parallelism in Algorithms and Architectures* (SPAA 08), ACM Press, 2008, pp. 217-228.
8. D. Mehta ed., *Handbook of Data Structures and Applications,* Chapman and Hall, 2004.

Arvind is the Johnson Professor of Computer Science and Engineering at the Massachusetts Institute of Technology. His current research focus is on enabling rapid development of embedded systems. He is coauthor of *Implicit Parallel Programming in pH.* Arvind has a PhD in computer science from the University of Minnesota. He is a Fellow of both IEEE and ACM, and a member of the National Academy of Engineering.

David I. August is an associate professor in the Department of Computer Science at Princeton University, where he directs the Liberty Research Group, which aims to solve the multicore problem. August has a PhD in electrical engineering from the University of Illinois at Urbana-Champaign. He is a member of IEEE and ACM.

Keshav Pingali is a professor in the Department of Computer Science at the University of Texas at Austin, where he holds the W.A.''Tex'' Moncrief Chair of Grid and Distributed Computing in the Institute for Computational Engineering and Science. His research interests include programming languages, compilers, and high-performance computing. Pingali has an ScD in electrical engineering from the Massachusetts Institute of Technology. He is a Fellow of IEEE and of the American Association for the Advancement of Science (AAS).

Resit Sendag is an associate professor in the Department of Electrical, Computer, and Biomedical Engineering at the University of Rhode Island, Kingston. His research interests include high-performance computer architecture, memory systems performance issues, and parallel computing. Sendag has a PhD in electrical and computer engineering from the University of Minnesota, Minneapolis. He is a member of IEEE and the IEEE Computer Society.

Derek Chiou is an assistant professor in the Electrical and Computer Engineering Department at the University of Texas at Austin. His research interests include computer system simulation, computer architecture, parallel computer architecture, and Internet router architecture. Chiou has a PhD in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a senior member of IEEE and a member of ACM.

Joshua J. Yi is a first year JD student at the University of Texas School of Law. His research interests include high-performance computer architecture, performance methodology, and deep submicron effects. Yi has a PhD in electrical engineering from the University of Minnesota, Minneapolis. He is a member of IEEE and the IEEE Computer Society.

Direct questions and comments to Arvind at the Stata Center, MIT, 32 Vassar St., 32-G866, Cambridge, MA 02139; arvind@csail.mit.edu.