usenix ;login:

OCTOBER 2012 VOL. 37, NO. 5

ELECTRONIC SUPPLEMENT

Conference Reports

4th USENIX Workshop on Hot Topics in Parallelism (HotPar '12)

Berkeley, CA
June 7-9, 2012

Surfing in Tandem: Parallelism and the Web

Summarized by Matt Sinclair (mdsincl2@illinois.edu)

Parallel Programming for the Web

Stephan Herhut, Richard L. Hudson, Tatiana Shpeisman, and Jaswanth Sreeram, Intel Labs

Dr. Stephan Herhut began the HotPar '12 workshop with a discussion of how to enable more parallel programming for Web applications, which are commonly programmed in JavaScript. His talk introduced River Trail (https://github.com/RiverTrail/RiverTrail), an open source system which makes expressing parallelism in JavaScript programs easy and doesn't require programmers to write parallel code. The main components of this talk were the design choices they made for River Trail and how River Trail enables JavaScript programs to be run in parallel.

As JavaScript is written predominantly by amateurs and beginner programmers, Herhut highlighted the importance of making River Trail easy to use and of taking sequential code written by others and using it in a parallel context. To this end, they made sure that all of their programs are still correct sequential JavaScript programs, albeit ones that are safe for parallel execution. Additionally, it needs to be portable and able to execute with high performance on *both* sequential and parallel architectures, which requires the runtime to exploit the available parallelism on each architecture. To make this practical, they decided to use high-level parallel patterns and have the runtime decide on how to best run the program. River Trail maintains the key safety, security, and deterministic execution tenets of JavaScript by using a fully managed runtime and disallowing side-effects on shared (immutable) state during parallel execution.

River Trail's API uses a data-parallel model and compiles the JavaScript programs for various parallel hardware to utilize the vector units and/or available parallelism. The API uses a three pillar approach. First, they introduce a new data structure, ParallelArray, which is immutable, dense, and homogeneous; these traits help obtain good performance and simplify the implementation and performance reasoning. Second, they introduce six methods—map, combine, reduce, scan, filter, and scatter—which Herhut claimed are sufficient to implement most use cases. Finally, they use elemental functions written purely in JavaScript and free of side effects; these functions require a trust-but-verify contract with the programmer, and River Trail verifies that these functions are indeed side-effect free.

After discussing these concepts, Herhut showed how River Trail operates at runtime. When a programmer calls one of their six methods, River Trail intercepts the call and runs its compiler instead. Next, they use the Intel OpenCL SDK to generate binary code. After the OpenCL runtime executes the code and the result is ready (they chose OpenCL because it can be run on many hardware platforms), they inject it back into the JavaScript program and execution continues. For all other calls in a program, the JavaScript executes normally. Herhut showed that, for a particle physics and a matrix multiplication benchmark, they are able to obtain speedups over the default JavaScript implementation and over a sequential C implementation of matrix multiply (to demonstrate that their speedups aren't just inflated by comparing them to JavaScript, which is quite slow).

There were numerous questions on the various components of River Trail. First, there were several questions about the compiler aspects of River Trail. Burton Smith (Microsoft Research) asked if it was strict or lenient with respect to function calls. Herhut replied that it is strict. Hans-J. Boehm (HP Labs) asked how side effects are handled, in particular if writing to the heap is allowed. Herhut responded that River Trail's compiler statically detects if there are writes to the global shared heap, but that this isn't always sufficient, so the runtime also has to check for it. Luis Ceze (University of Washington) asked if the authors had looked into auto-parallelizing JavaScript. Herhut replied that doing this for existing code is really hard, and JavaScript makes it even harder because its semantics don't list what the side effects are (the second presenter explains that JavaScript's DOM sequentializes things, which creates lots of synchronization costs) and makes it unclear what to access. Tracking state is one possibility, but doing this would create overheads and eat into the benefits they see from this approach.

Second, there were questions on the benchmarks they used. Paul McKenney (IBM Beaverton) asked why they weren't able to get single-threaded results for their prototype (all single-threaded JavaScript results presented were the default JavaScript results). Herhut responded that the OpenCL implementation can't run single-threaded, but that if it could, they expect to still see some performance gains because their JIT is more efficient since typical JIT engines don't know whether there's a benefit to doing things like unrolling, while River Trail's JIT does know this. Luis Ceze asked if they had looked into any other applications. Herhut said that they had also looked into real-time physics for game engines, image processing, and modification benchmarks, and that they'd also seen some speedups there.

There were also several questions about parallelism they obtain from JavaScript. Burton Smith noted that here the parallelism is abstracted away from the hardware resources fairly well, and was curious how they decided how many hardware resources to apply. Herhut responded that they could try to grab every resource, but that chose to leave that out of the programming model and let runtime deal with it. As a result, the number of resources used is client dependent. This is something they are still working on though, Herhut concluded.

Finally, several audience members pointed out that WebGL provides a way to stick kernels into code with some of the CUDA/OpenCL features, which game developers are starting to exploit for optimized browser execution; the theme of their questions was how River Trail copes with coexisting highly parallel programming models. Herhut admitted that they don't yet have a good answer for this problem, because this would require a centralized management of resources that doesn't exist yet. Sam King (University of Illinois at Urbana-Champaign) pointed out that this is a fundamental flaw of modern OSes, not just these projects, which provided an interesting tie-in to the second session of talks.

A Case for Parallelizing Web Pages

Haohui Mai, Shuo Tang, and Samuel T. King, University of Illinois at Urbana-Champaign and Valkyrie Computer Systems; Calin Cascaval and Pablo Montesinos, Qualcomm Research

Time is money for companies such as Google and Microsoft. Unfortunately, Web browsing on mobile devices is very slow, which Google estimates costs them approximately $200M per year in revenue (for a 0.7% increase in delay of searches). The client CPU presents a significant bottleneck in attaining high performance Web browsing on mobile devices. Mobile CPUs are limited in the performance they can attain by device form factors and battery life. With the arrival of multicore mobile devices, parallelizing Web browsing would appear to be an attractive option; however, the current structure of modern Web programming is inherently sequential, which makes this difficult. In this talk, Haohui Mai introduced Adrenaline, a system that attempts to speed up Web apps for multicore mobile devices.

Adrenaline consists of two components: a server-side preprocessor and a client browser on the mobile device. When a user accesses a Web page, the request is sent to the server. At the server, the Web page request is decomposed into "mini-pages," which can be processed in parallel and sent back to the browser as they're completed (the server attempts to aggregate the mini-page responses so that the user sees them all appearing simultaneously). Finally, the browser on the mobile device is responsible for putting all the mini-pages back together in a coherent manner (such that the Web page looks exactly like it would without using Adrenaline) and presenting the page to the user. This approach improves performance, maintains the current Web applications semantics, and requires minimal modifications, all of which help to encourage adoption.

Maintaining the current Web application semantics introduces several important challenges. The HTML for a Web page says what should appear, and internally the browser parses the HTML into a DOM, the core data structure used by many browser components such as JavaScript, which takes the shape of a tree. Identifying places in the DOM that are suitable for parallelization is crucial in Adrenaline. Additionally, each of the mini-pages needs to be compatible—the Web developer shouldn't need to change anything; they should only see their performance improved. To ensure this compatibility, while they are distributing the DOM inside Adrenaline, they maintain a main page that only executes the JavaScript. All created mini-pages are merged back together on the main page. To reduce overhead, they pre-compute and cache results. The use of a Bloom filter minimizes synchronization overhead when accessing the pre-computed results.

The Adrenaline browser is built on top of QtWebKit. To demonstrate its power, they evaluated it on 170 of the most popular Alexa Web sites using a quad-core ARM board with the Web sites being run locally to ensure repeatable results. Mai showed that Adrenaline is able to achieve a mean speedup of 1.54x and a mean latency reduction of 1.75 seconds. Overall, Adrenaline improved performance for 89% of the 170 sites. They also performed a case study loading the Nokia Wikipedia page. When the page loaded on Adrenaline instead of QtBrowser, latency was reduced by 12 seconds and they achieved a 3.34x speedup. A breakdown of their speedups showed that they incur some synchronization overhead, but that the improvement from parallelism is larger. Additionally, they see that they need to perform less work in Adrenaline because of the decomposition, which creates smaller working sets. Finally, Mai showed a demo (http://www.youtube.com/watch?v=TBcurpe89PI) where Adrenaline was noticeably faster than QtBrowser on a Tegra 2.

Mai received numerous questions at the end of his talk. Luis Ceze (University of Washington) asked him why they can't do the decomposition in the browser itself, because the server seems like a liability. Ceze added that locally caching and reusing seemed like a better idea. Mai replied that they observed that the mobile Web browser is serial-bound, so pushing more stuff onto the mobile device would cause a bottleneck due to its limited resources.

David Johnston (Canon Research/CiSRA) asked if Adrenaline Web pages can have fancy layouts and rendering. Mai responded that they implemented mini-pages as a plugin, so it's a process and the content is in a Web browser, so everything that a Web browser can lay out are things it can do.

Nicholas Matsakis (Mozilla Research) asked how sensitive their performance results are to the connection to the server. Mai replied that Adrenaline works like HTTP processing: get page, process it; things like images may go through a direct connection between remote server and client. They could do this differently, though, which is ongoing work. Matsakis later asked how flexible Adrenaline's decomposition is, how much work the server does, and whether there's a floating drive that affects the layout. Mai replied that their current implementation renders the whole page on the server, figures out the layout, then determines what's appropriate to move to mini-pages.

Stephan Herhut (Intel Research) asked, since pages are rendered once then looked at, is there a benefit for dynamic pages? Mai answered that this depends on the characteristics of the Web applications. This isn't fundamental to their approach, but it is how they currently do it. David Reed (SAP Research) asked if the added network latency offsets the benefits they see in Adrenaline by parallelizing. While they do pay a price for more network requests, Mai said, they cache everything locally, which ignores some of these effects. When they have run non-local experiments, they haven't seen this become a problem yet.

Paul McKenney (IBM Beaverton) asked if some of Adrenaline's benefits come from the server caching, then parceling out the data to multiple users. Mai replied that this was partially true, and that they don't do well for heavily personalized pages (like Facebook) because they have to do decomposition every time. Finally, Burton Smith (Microsoft Research) asked how they determine how many hardware resources to apply. Mai replied that this is future work, but that they haven't worked on it much yet.

We Need Support! OS Support

Summarized by Matt Sinclair (mdsincl2@illinois.edu)

For Extreme Parallelism, Your OS Is Sooooo Last-Millennium

Rob Knauerhase, Romain Cledat, and Justin Teller, Intel Labs

The design philosophy of OSes has worked well for current systems because programmers didn't need to worry about the hardware details, instead relying on hardware to continue improving their performance. Unfortunately, at exascale levels, this isn't true, largely because the cost of data movement is an overriding concern. Additionally, current OSes don't deal with communication costs. At exascale, the systems will require an unprecedented amount of complexity in managing the effective coordination and use of resources. Unfortunately, according to Romain Cledat, many of the traditional OS functions are not optimal for exascale, which leads him to propose eliminating the traditional OS.

In their proposed exascale system (Runnemede), instead of a traditional OS and traditional cores, they instead have "Control Engines" (CEs) and "Execution Engines" (XEs) arranged in a hierarchy. The CEs, which are sprinkled throughout the system, are responsible for executing the runtime environment and do not execute (directly) user code, while the XEs, which are much more numerous, are responsible for executing the application code only. Such a separation allows them to specialize the duties and hardware in each core. The system breaks programs into codelets, which are small chunks of code with dataflow-like dependencies between them. Codelets run uninterrupted on the XEs and fire only when their dependencies are met. The system software for such a system needs to be able to dynamically adapt and move data around (which current OSes can't do). Given this baseline system, Cledat discussed three proposed changes to the OS for increased energy efficiency: separation of concerns, memory management, and threading abstractions.

Traditionally, resources are shared between a kernel that interacts with the hardware and user code. However, the latency of switching between user and kernel mode is too expensive in terms of energy and latency for exascale. In Runnemede, since resources are plentiful at exascale, they remove this distinction and instead provide separate privileges spatially instead of temporally—the CEs run the kernel code and the XEs run the user code. This allows for specialization of resources (queue processing on CEs, energy efficiency and custom functionality on XEs). Additionally, since system code only runs on the CEs, there is no need for device drives in the traditional sense.

In traditional OSes, where virtual memory abstracts away the physical memory, memory management overhead is too high for exascale, especially in terms of loss of visibility into the memory hierarchy—exploiting locality as much as possible is crucial. Simply making virtual memory smarter is not sufficient, because it uses the granularity of pages, which isn't well suited to multiple levels of memory hierarchy (which an exascale machine will likely have). Additionally, the OS's support for memory allocation is also agnostic to the characteristics of the underlying memory, as they are unable to efficiently move around since its access pattern varies over time. To solve these issues, the authors propose to make data a first-class object. A runtime system can track the usage of the data at a application-dependent controllable granularity. This enables memory management to be done in a way that minimizes wasted movement, while precisely placing data in memory and allowing it to be moved around as the access pattern changes over time.

Scheduling management in traditional OSes is also an issue. Context switching is very expensive in terms of latency and energy—exascale machines will want to avoid it as much as possible since it's trying to "save" a plentiful computing resource. Additionally, Cledat argued that exascale machines will want to avoid hard-binding to hardware resources because the tradeoffs in the system will change during an application's lifetime. In short, in exascale systems, co-locating the thread and its data is key to reducing energy. Because XEs are plentiful in their system, and because they use codelets, Cledat's solution to this problem is to expose the affinity interface among tasks and data instead of to particular hardware or threads. This allows the runtime to adapt to the time-varying capabilities of the underlying hardware and provides an explicit dependency interface. By making the dependency information available at the lowest levels of the system software, they can make better scheduling decisions.

When asked how such a system would be programmed since he advocates removing the abstraction of a big global address space, Cledat responded that they aren't proposing to remove the address space, but that they just want visibility into it and some contract about what will happen. Further questions on this related to how to specify where data will be placed, such as using instructions to specify where to put data. Cledat replied that the audience should think of this as happening in a NUMA fashion, where there is memory physically close and far from a core, and the core wants to use the memory that's close to it as much as possible.

David Reed (SAP Research) asked if the authors had a specific design proposal for allowing an application to reflect on its own behavior, or how an application could take into account what power got spent where (i.e., a way to express causality). Cledat answered that they haven't implemented something like this but are interested in looking into it. Reed asked how this approach scales to large-scale memory systems/data management. Cledat replied that is another issue they haven't solved directly yet. So far, they've only looked at a decomposition approach where data is decomposed into smaller sets that go onto on-chip memories (i.e., dividing the data up into reasonable chunks). They're looking at things like Hierarchically Tiled Arrays (HTAs) to help, but this still leaves a missing component in the middle. Hans-J. Boehm (HP Labs) asked how the address of an object and its references are updated when the object is moved. Cledat responded that the object moves in the address space and updates all the references that use it. The compiler is responsible for generating proper instructions to decode the offset of the moved pointer. In this approach, they only update one field in each of the users and rely on indirection to help (similar to physical virtual memory addresses).

Session chair Sam King (University of Illinois at Urbana-Champaign) asked Cledat to compare the work being done at Intel with the work being done at Wisconsin, since they are advocating very different positions—the Wisconsin group is advocating adding functionality to the OS, while the Intel group is advocating removing functionality from the OS. Cledat responded that they are looking purely at exascale, while the Wisconsin project is looking at "smaller" platforms, where it's possible that adding more functionality to the OS is the correct approach.

Operating Systems Should Manage Accelerators

Sankaralingam Panneerselvam and Michael M. Swift, University of Wisconsin, Madison

In the past, the additional transistors available with each generation of CPU were better used improving general-purpose execution than accelerating a single computation, despite the efficiency advantage accelerators offered for some applications. However, with the rise of dark silicon, this approach is no longer viable because not all of the transistors can be on simultaneously. As a result, incorporating accelerators on-chip has become more attractive. However, there is little or no support in current OSes for accelerators. In this talk, Sankaralingam Panneerselvam proposes to solve that issue by managing the accelerators in the OS. Specifically, this work is focused on how the OS can support these diverse devices while abstracting heterogeneity (virtualization), enabling flexible task execution (task invocation), and multiplexing shared accelerators among applications (scheduling).

First, Panneerselvam presented a taxonomy of accelerators (based on Table 1 in the paper) and identified resource contention as a cross-cutting issue and access methods as a differentiator between the classes. All of the suggestions he made later in the talk were based on this taxonomy.

Next, Panneerselvam raised four challenges in programming accelerators. First, current systems decide statically at compile time where to execute a task. However, a task can be executed through different means (e.g., a parallel task can be run on a CPU or GPU). This static method fails to capture application factors like data granularity and system-level constraints like power limits. Second, virtual addresses used by programs must be translated physical addresses for an accelerator. Third, asynchronous accelerators may produce results after a task has been context-switched out. Finally, shared accelerators requires some sort of scheduling to prioritize accesses. This is particularly difficult for accelerators that can be directly accessed from user mode.

To combat these issues, Panneerselvam proposed a new OS design (Rinnegan) where calls to accelerators are treated as function calls that can be dynamically dispatched to an accelerator *or* executed locally on the CPU at runtime. The OS is responsible for enforcing the system-wide policies and tries to provide support for the accelerator classes. Rinnegan allows accelerators to be integrated into common programming paradigms with little effort *and* provides flexibility on when they're used. Effectively, this means that calls to accelerators are treated as non-blocking processor calls.

The key components of Rinnegan are the accelerator stub, agent, and monitor. The accelerator stub is the entry point for invoking an accelerator; it abstracts different processing elements from the programmer and provides a single procedural interface to the application. The stubs are responsible for deciding where to execute the code, passing it and the data there, then implementing synchronization mechanisms or blocking if necessary. An accelerator agent acts as a middleman; its job is to manage the accelerator by providing mechanisms to bind programs to accelerators, creating communication channels to expose accelerator usage to the OS to guide policy decisions and to implement scheduling decisions for the accelerator based on OS policy. The accelerator agent is the local scheduler in a two-level scheduler. Finally, the accelerator monitor is a system-wide service that monitors utilization of all the resources in the system. In a two-level scheduler, the monitor plays the role of global scheduler, maintaining system-wide constraints such as minimizing power or maximizing performance (the agents are the local scheduler).

Panneerselvam was asked if his work had any notion of data-driven task invocation, like the codelets in the Knauerhase paper. Panneerselvam answered that this isn't something they have focused on yet but would like to in the future. David Reed (SAP Research) asked if they are looking at the composition of accelerators. Panneerselvam responded that they are, because while every core has support for doing AES, obviously a cryptographic accelerator is best-suited for this, and that one way to choose between them is the data granularity: depending on size, one may want to run it on different devices in the system.

Paul McKenney (IBM Beaverton) asked if hardware will evolve to the point that there's an accelerator associated with each core (like vector units) and, if so, whether this will affect the approach discussed in this talk. Panneerselvam replied that a tradeoff should be made between performance and flexibility of the accelerator units while designing processor chips. OS involvement might still be needed to maintain constraints such as power limits. Burton Smith (Microsoft Research) talked about how hardware vendors are implementing shared virtual memory, such as AMD's Fusion, and asked what Panneerselvam thought was a good way to implement shared virtual memory in this space (perhaps having TLBs for every accelerator, determining how they're shut down, how to do demand paging, etc.). Panneerselvam didn't have a specific answer for this question, but talked about some possibilities.

David Johnston (Canon Research/CiSRA) followed up by asking if a "super MMU" could replace the accelerator stubs and other components in the proposed design. Panneerselvam replied that smarter MMUs could definitely be of great help, but that the accelerator agents are also responsible for other functions like local scheduling and exposing usage. Finally, David Reed asked if this approach would scale to large-scale memory systems in terms of data management. Panneerselvam replied that this might be a new category of problem for the OS. As with the previous paper in this session, Sam King asked Panneerselvam to compare the work being done at Intel with the work being done at Wisconsin. Panneerselvam agreed with Cledat; because the Wisconsin project looks at "smaller" platforms, adding functionality to the OS could be the correct approach.

Discipline Action: Programming Models

Summarized by: Zoltan Majo (zoltan.majo@inf.ethz.ch)

Parallel Closures: A New Twist on an Old Idea

Nicholas D. Matsakis, Mozilla Research

Nicholas Matsakis describes a framework for parallel programming that statically guarantees data-race freedom. The framework uses closures to express parallelism. These closures can be executed in parallel with their parent task (i.e., the code that declares and schedules them). However, there is a problem with using closures to define parallel tasks: programs that use closures are prone to contain data races, as closures and their parent task can be executed in parallel and thus they can simultaneously access shared data. Nicholas' work presents a solution to this problem based on two changes to parallel closures.

The first change involves the schedule of parallel closures. In Nicholas' system a parent task and its children are scheduled one after another, and therefore the parent and its children are guaranteed not to access shared data at the same time. The first change eliminates data races between the parent and its children, but children can be executed in parallel, and thus they can still cause data races. The second change involves changing the type system so that it guarantees that state data is read-only. More specifically, a child is not allowed to modify data found in its surrounding environment nor data obtained from a sibling closure.

Making all data available to parallel read-only is too restrictive, so the framework presented by Nicholas provides a way to control accesses to mutable data. One example presented in the talk involves offering different views of a shared array to closures using the array.

Russell Williams asked whether there is anything special to this approach that is not already available in C++. Nicholas replied that casting is problematic in C++, but otherwise the library is not much different. Vivek Sarkar asked whether it is possible to use static fields. Nicholas said yes. Sankaralingam Panneerselvam asked about the difference between the join() and get() methods. Nicholas said that get() is used by the parent task, join() is used by sibling closures. Paul E. McKenney wondered whether async was necessary. Nicholas answered that it is, otherwise a parent could mutate data after tasks have been created.

"Simultaneous" Considered Harmful: Modular Parallelism (Deprecating "Locks," "Semaphores," Serializability, and Other Sequential Thinking)

David P. Reed, SAP Research

David Reed stated that the world is embarrassingly parallel, and yet it still works without using globally synchronized clocks. Parallel programming is now widespread, but we parallel programmers consider it more as an acceleration of sequential programming. As a result, our way of thinking about parallel programming is overly restricted, and this limits our ability to efficiently use parallel machines. The author proposes that we should think of parallel computing as the norm, and we should consider sequentializing computations to be the hard problem.

David proposed several calls to action: (1) the scope of parallel action should be limited, because simultaneous action at a distance is a bad habit; (2) serializability is probably not the correct definition of correctness; (3) good modularity should never rely on the concept of simultaneity; (4) programmer thinking should be fixed by teaching parallel programming first; and (5) Amdahl's Law should be rejected, because it dominated only because programs are conceived as sequential, and not because problems are sequential.

As practical examples of programming constructs that are problematic in today's parallel programming David listed semaphores, the mprotect() and open() calls, and compare-and-swap operations. Instead of these constructs, clean primitives should be used and should include but not be limited to the following: fork() and join(), event counts and sequencers, producer-consumer LIFO and FIFO buffers, and write-once and read-many memory cells.

Disciplined Concurrent Programming Using Tasks with Effects

Stephen Heumann and Vikram Adve, University of Illinois at Urbana-Champaign

Stephen Heumann stated the case for a task-based parallel programming model that offers strong correctness guarantees based on programmer-specified effects. The model targets programs that rely on parallelism not only to speed up computations, but also to guarantee interactivity (e.g., GUIs). Stephen defines strong correctness guarantees as data-race freedom, strong atomicity, deadlock freedom, and deterministic semantics. Current parallel programming models are limited either because they offer restricted forms of parallelism or have high overheads, or in some cases both.

The model proposed by the authors plans to address these problem by requiring parallel programmers to annotate each task of a parallel program with effects. Effects specify for each task the memory regions read/written by the task. The compiler statically checks effect summaries; thus there is no need to check effects at runtime. Based on the computed effects, the runtime system schedules tasks so that tasks having overlapping effects do not execute concurrently. As a result, the model guarantees data-race freedom.

An interesting aspect of the model is effect transfer. On task creation (implemented by the spawn() operation), the effects of the parent are transferred to the created task. When the created child task is joined, the effects of the child task are transferred back to the parent. The model does not guarantee deadlock freedom, but effect transfers help to avoid deadlocks possible because of the effect system.

Someone asked about tasks whose effects are not known statically. Stephen answered that there can be a runtime component that checks effects at run time. Additionally, the system could be extended to support types parameterized with regions and effects. Someone else asked what happens if a task is annotated improperly (e.g., task modifies everything); in this case, the static analysis would probably fail. Stephen replied that the system could deduce at runtime what the actual effects are (using the previously mentioned dynamic-effect checking).

Bringing Down the House: Speculation

Summarized by Brandon Myers (bdmyers@cs.washington.edu)

HydraVM: Extracting Parallelism from Legacy Sequential Code Using STM

Mohamed M. Saad, Mohamed Mohamedin, and Binoy Ravindran, Virginia Tech

Mohamed Saad presented work on automatically parallelizing general sequential code using software transactional memory (STM). One motivation for this work is that very large legacy code bases tend to consist of mostly sequential code (for development cost and scalability reasons). Sometimes source-level transformations or manual refactoring is not possible because the source code is unavailable. Automated concurrency refactoring at the bytecode level can address this problem.

HydraVM is a speculative parallelization system that works by analyzing the program statically and with dynamic profiling, constructing superblocks that may be good to be run in parallel, using them to reconstruct the program into a producer-consumer pattern, and executing the blocks as transactions. Superblocks are formed by representing a sequence of executed code as a string and performing string factorization. Transactions may commit in program order (1) if the block was actually reachable and (2) if there were no memory conflicts with previous blocks. Excessive commit conflicts can hurt performance. HydraVM continuously collects conflict information to tune the superblocks; conflicting superblocks are coalesced and recompiled.

HydraVM parallelization shows 2–5x speedup on sequential implementations of JOlden benchmark algorithms, which exhibit data-level parallelism. The source code is available at www.hydravm.org.

Parallelization by Simulated Tunneling

Amos Waterland, Harvard University; Jonathan Appavoo, Boston University; Margo Seltzer, Harvard University

Amos Waterland presented "basic research" towards an aggressive goal of a different kind of computing system that uses a generalization of memoization that is based on "dynamical systems." It consists of a master compute node and a huge "intelligent cache" made up of a large cluster of predictive simulation nodes. The master always knows the true current state of the program and evolves the state. Each cluster node simulates an x86 processor starting from some prior distribution over program states, finding trajectories to proceeding states. The master advances the program by first querying all predictor nodes, asking if they have seen its current state before. If there is a match, then the master node can "tunnel" to the endpoint state of the trajectory that has been calculated from that start state. The master node has skipped ahead in the program using previously computed results of the predictor node. There are many predictor nodes working on different parts of the state space, so the system is performing parallel execution of a sequential program.

Poster Session and Reception

Summarized by Dongdong Deng (deng@cae.wisc.edu) and Brandon Holt

Performance Implications of Co-Scheduling Modern Parallel Applications on NUMA Multi-Core Systems

Cheol-Ho Hong and Chuck Yoo, Korea University

Cheol-Ho Hong presented their investigation on the performance impact of co-scheduling parallel threads in the NUMA system. The NUMA system has increasingly attracted researchers' attention due to its capability of overcoming the limitation on the number of cores. AMD and Intel have applied cross-chip interconnect technologies to their products where one processor can access remote memory belonging to other processors via a cross-chip connect. However, a recent study has found that no significant performance improvement can be obtained on PARSEC by gathering threads in the same cache. Thus, Hong and Yoo studied the impact of co-scheduling and found that the performance of parallel threads is highly dependent on the miss rate of the last level cache as well as the pattern of memory usage in each thread.

Evaluation of Hardware Synchronization Support of the SCC Many-Core Processor.

Pablo Reble, Stefan Lankes, Florian Zeitz, and Thomas Bemmerl, RWTH Aachen University

Pablo Reble presented work on hardware support of fast synchronization methods. The authors believe that hardware support is critical to achieving high performance. An increasingly large number of cores have been integrated into one chip, resulting in degraded performance. What's more, the problem of coherence becomes severe as the number of cores increases due to the performance overhead introduced by hardware cache coherency protocols. In addition, software-based coherency has become an alternative way for shared memory programming models on manycore systems. Thus, the authors studied the characteristics of the hardware synchronization support of a cluster-on-chip architecture.

Does Shared-Memory, Highly Multi-Threaded, Single-Application Scale on Many-Cores?

Ghassan Almaless and Franck Wajsburt, UPMC Sorbonne Universités

Ghassan Almaless presented work on the study of scaling highly multithreaded applications on manycore systems. Manycore systems can now support up to 100 cores, and it is reasonable to believe that hundreds of cores can be integrated into one chip in the near future. The cores have been designed increasingly simple with small-sized cache in consideration of power efficiency. In this study, the authors found the scalability limitation of SPLASH-2 FFT and EPFilter on a simulated 512 core system. Their analysis shows that the scalability limitation not only depends on the notion of thread and process, but also becomes worse due to the small cache of each core.

A Lightweight Approach to Compiling and Scheduling Highly Dynamic Parallel Programs

Ettore Speziale and Michele Tartara, Politecnico di Milano

Ettore Speziale presented work concerned with a dynamic and lightweight compiler that is able to control the execution of highly dynamic multithreaded programs at runtime. With this dynamic and lightweight compiler, a full-fledged just-in-time compile is no longer needed. In addition, this lightweight compiler is capable of performing runtime optimizations that are impossible to conduct during the compilation due to lack of sufficient information. Furthermore, the optimizations can be conducted multiple times according to the actual runtime environment, which confines the performance overhead to a negligible amount.

ExM: High Level Dataflow Programming for Extreme-Scale Systems

Timothy G. Armstrong, University of Chicago; Justin M. Wozniak, Michael Wilde, and Ketan Maheshwari, Argonne National Laboratory; Daniel S. Katz, University of Chicago and Argonne National Laboratory; Matei Ripeanu, University of British Columbia; Ewing L. Lusk, Argonne National Laboratory; Ian T. Foster, University of Chicago and Argonne National Laboratory

Timothy G. Armstrong presented work related to a high-level data flow programming model for extreme-scale systems. They propose an extreme-scale many-task (ExM) programming and execution model, overcoming the difficulties of higher-level logic of complicated multithreaded applications on large scale systems. ExM offers an high-level and demonstrative programming model that enables pervasive parallelism by accurate and automated concurrent execution. Furthermore, ExM includes integration of dataflow constructs, function evaluation, and task generation, solving the challenges of programmability and scalability required by highly paralleled systems. Finally, Armstrong demonstrates the potential applications of current ExM implementation.

They are currently restricting the work to deterministic executions, and all I/O must be done at the beginning of the program so that there is a starting state. The state is represented by one vector of the memory in the machine. The state is evolved with an evolution rule that implements something like one x86 fetch-decode-execute. This is a dynamical system. The master node's state vector is matched if a predictor node has a vector that is equivalent under a symmetry transform. Currently, translation is the symmetry used for the x86 dynamical system. That is, if certain memory bits did not affect a trajectory, then those do not have to match.

A predictor node starts with some prior distribution of starting program states. It can either spend cycles advancing a trajectory or exploring new possible starting states based on its belief distribution.

Luis Ceze commented that the technique is like memoization except at a granularity that is not necessarily intuitive to a human. Amos replied that, yes, it is like memoization but more general. He imagines that generations of a civilization could pass down accumulated knowledge in the form of important computed trajectories. Luis also referred the attendees to related work, "Massively Speculative Parallelization" from MICRO 2003.

On the subject of reuse, David Reed asked how the "don't care bits" for translation symmetries are inferred. Amos replied that they keep track of which bits are used in an application of the evolution rule, and those not used are assumed to be non-causal. Symmetries play a crucial role; if only 5% of the bits in the state vector have to match, the prediction problem gets easier and the trajectory cache is more useful.

Burton Smith made a related comment regarding memoization techniques, that sometimes higher level knowledge can reduce the distinguishing state to a single integer. He also suggested it would be possible to use more than one master compute node, each starting at different places. Amos said that knowledge can be injected into the cache. Luis mentioned that in some applications and systems there is massive reuse, such as shared structure in cloud computing.

Luis asked how we should consider speculation in a world concerned about energy. Mohamed agreed that it is important to study performance and energy tradeoffs. Amos replied that right now they use full CPUs for everything (32,000 cores of IBM Blue Gene at Argonne Lab as part of their DOE grant) but that in a full realization of the tunneling computer, the different pieces should be specialized to achieve efficiency. The predictive inference could use low-power neuromorphic devices, the parallel cache search could use a large parallel associative memory, and only the speculative execution would require a full CPU.

Amos also presented his work at a physics conference, suggesting applications in molecular dynamics. In future work, he would like to explore other possible symmetries; in the x86 tunneling system they only use a form of translation.

Poster Session

CnC-Python: Multicore Programming with High Productivity

Shams Imam and Vivek Sarkar, Rice University

Shams Imam (shams@rice.edu) from Rice University, advised by Vivek Sarkar, presented this work which aims to enable Python programmers to leverage parallelism in their applications using the Intel Concurrent Collections (CnC) paradigm. CnC is a declarative parallel coordination model where computation is expressed in terms of serial "steps" and declarations of data and control dependences. CnC aims to allow non-expert programmers to be productive writing efficient parallel programs, so Python is a natural implementation choice. One of the primary challenges in the implementation was that CPython, the most prevalent implementation of Python, has a global interpreter lock which makes running multiple threads in parallel difficult. Using other popular Python implementations which do support parallelism is suboptimal because they are incompatible with many extension modules. To overcome this obstacle, the CnC-Python library calls out to the Habanero Java implementation of CnC, which is multithreaded, which then schedules user-specified Python code to run safely across those threads.

Gains in Conjugate Gradient Computation with Linearly Connected GPU Multiprocessors

Stephen J. Tarsa, Tsung-han Lin, and H.T. Kung, Harvard University

This work out of Harvard focuses on efforts to accelerate conjugate gradient computation on graphics processors to allow for more efficient processing of compressive sensing signal reconstruction. On the surface, this problem has abundant, regular, data parallelism, so it should be highly amenable to GPGPU acceleration. However, each stage of the iterative process involves a global vector normalization step, which on current generations of GPUs cannot be done without an expensive synchronization, either through long-latency communication through the host CPU or global GPU memory. In order to mitigate this bottleneck, the authors propose a linear series of connections between stream multiprocessors (SMs) which would allow a 15x improvement in their implementation. They believe that the overhead of introducing such a small change to the architecture should not be very expensive, but they admitted that it would be difficult to integrate such a primitive interconnect cleanly in something like the CUDA-C programming model because the scheduling of thread blocks is completely opaque.

Concurrent Predicates: Finding and Fixing the Root Cause of Concurrency Violations

Justin E. Gottschlich, Gilles A. Pokam, and Cristiano L. Pereira, Intel Corporation

Debugging concurrency violations is notoriously difficult because their non-deterministic nature makes them difficult to recreate consistently. The *concurrent predicates* (CPs) coined in this work allow a programmer to single out a bug by specifying the conditions that are known to trigger the bug. In a live demo of the tool, Justin Gottschlich (justin.e.gottschlich@intel.com) showed how these CPs, which look like simple function calls, can be inserted into the buggy code at one of the points where the bug is known to reside. When executed, we could see how the CPs delayed threads' execution so that they consistently recreated the particular interleaving known to cause the bug. Justin also demonstrated more parts of the tools that allow for some automation in placing predicates, although he warned that this introduces a risk of false-positives appearing.

Middleware for Many-Cores—Why It Is Needed and What Functionality It Should Provide

Randolf Rotta, Steffen Büchner, and Jörg Nolte, Brandenburg University of Technology

This work consists of an exploration of manycore platforms such as Intel Knight's Corner, AMD Interlagos, or Tilera TilePro100. These platforms, while having much in common architecturally, provide differing coherence and communication mechanisms, which, while they provide a common interface, have drastically different performance implications, obviating the need for middleware to make software written for these platforms more performance-portable. Some of the useful functionality includes: support for cross-core method calls and ways to coordinate groups of tasks. The authors of this work ported the existing TACO template library to use shared memory rather than MPI and ran several benchmarks to understand the performance implications of their design.

The Road to Parallelism Leads Through Sequential Programming

Gagan Gupta, Srinath Sridharan, and Gurindar S. Sohi, University of Wisconsin—Madison

This work proposes a parallel programming model that, while asking the programmer to explicitly express parallel regions, enforces serial semantics to help programmers reason about their program. The system, described in previous publications, dynamically schedules non-interfering tasks (as specified by explicit read/write sets) to run in parallel. This work focuses on automatically tuning parallelism to improve efficiency and a way to implement low-overhead restartable parallel execution. Using a "Goodness of Parallelism" (GoP) metric composed of periodic samplings of efficiency, the authors determine whether excessive contention is hurting efficiency, making it beneficial to use less parallelism. This approach provides many of the benefits claimed by STM and other speculative parallelism approaches with lower overhead but, as a tradeoff, requires more programmer knowledge to express the read/write sets.

Elastic Scaling for Transactional Memory: From Centralized to Distributed Architectures

D. Didona, Instituto Superior Técnico/INESC-ID, Portugal; P. Felber and D. Harmanci, University of Neuchâtel, Switzerland; P. Romano, Instituto Superior Técnico/INESC-ID, Portugal; J. Schenker, University of Neuchâtel, Switzerland

Transactional memory allows threads to safely run in parallel. However, the more threads that are operating at once, the more likely they are to collide and cause each other to be squashed. Because the likelihood of overlap can be highly program-, phase-, and data-dependent, the optimal number of parallel threads must be determined dynamically. This work explores and contrasts dynamic parallelism tuning on shared and distributed memory machines. For shared memory machines, the cost of being wrong is not too high, so it is sufficient to make some measurements to characterize the workload and do simple hill-climbing to find a good number of transactions. In a distributed setting, many more factors interact interdependently and non-linearly, and the cost of being wrong is much higher. They show that in these cases, it is necessary for the runtime to construct a model of the performance characteristics and simulate transaction-scaling decisions to choose the best one.

SMT QoS: Hardware Prototyping of Thread-Level Performance Differentiation Mechanisms

Andrew Herdrich, Ramesh Illikkal, Ravi Iyer, Ronak Singhal, Matt Merten, and Martin Dixon, Intel Corporation

Simultaneous multithreading (SMT) allows multiple independent threads to run in the same pipeline. However, some operations, particularly memory operations which missed the cache, take significantly longer to complete than others. When two SMT threads have differing workloads (i.e., one has more memory operations), that thread is unable to use its pipeline slots, and that can cripple the performance of the other compute-bound thread by tying up resources it could use to run faster. In addition to demonstrating this issue on the Nehalem architecture, this work also introduces and explores new forms of source-based execution rate control at the pipeline level to enhance SMT control. By limiting the instruction issue rate and limiting the amount of reservation station slots allocated for memory-bound threads, they show they can prevent useless starvation of compute-bound threads and improve overall efficiency.

Imposing Order: Scheduling

Summarized by Dongdong Deng (deng@cae.wisc.edu)

Fine-Grained Resource Sharing for Concurrent GPGPU Kernels

Chris Gregg, Jonathan Dorn, Kim Hazelwood, and Kevin Skadron, University of Virginia

Chris Gregg presented KernelMerge, a kernel scheduler enabling multiple applications to run concurrently on one device. Load balancing and resource allocation are two main problems. Allocating computing units and memory reasonably among computing-bound and memory-bound applications is critical to performance improvement. In this presentation, Gregg shows the scheduling algorithms, experiment results, limitations of KernelMerge, and a case study of determining optimal workgroup balance.

KernelMerge adopts two different scheduling algorithms: a round-robin work-stealing algorithm and a fixed percentage of workgroups assignment algorithm. Experiment results show that 39% of concurrent kernel pairs obtain a speedup. Also, the results indicate that a naive way of scheduling kernels is harmful. However, KernelMerge has three limitations. First, the high utilization of registers limits the number of workgroups that can use one device at the same time. Second, the usage of shared memory by the scheduling kernel reduces the number of possible workgroups that concurrently run on the same device. Third, barriers are not allowed if two kernels have different numbers of workgroups to avoid deadlock.

Finally, Gregg demonstrated the case study of finding an optimal workgroup balance. The results confirm that KernelMerge is effective in determining optimal workgroup and naive scheduling of kernels may have negative impact on performance. With KernelMerge, an 18% speedup is achieved for two concurrent kernels over the time to sequentially run these two kernels.

Do We Need a Crystal Ball for Task Migration?

Brandon Myers and Brandon Holt, University of Washington

Brandon Myers presented the authors' exploration of profitable task migration decisions based on data locality. To improve the performance of applications with intensive communication on distributed systems, network traffic needs to be reduced. Task migration is considered as a possible solution to achieve this. Myers presented their system model, simulation framework, online policies and evaluation results.

Myers showed that they use a simple model that merely focuses on the data amount transferred over the network. Thus, this model only has two layers of locality hierarchy: local and remote. Several simplifications have been made to obtain theoretical cost bounds in polynomial time. Also, load balancing is not considered in the model because local computation time is absent. With these constraints, the problem has been formulated as a single source, multiple destination shortest path problem over a DAG of task location over time.

The simulation framework includes two stages: one is generating a memory trace for each task in a multithreaded shared memory application and the other is simulating the sequence of memory accesses in the manner they happen on a distributed system. PIN, a binary instrumentation library, is used for collecting memory traces on annotated benchmarks. The simulator takes memory traces, an allocation table, number of distributed nodes, task size, and migration policy as inputs.

Hindsight Migrate policy is employed as a migration predictor. Unlike Stream Predictor, a task does not suffer from the cost of extra remote accesses to form a pattern. Instead, migration occurs immediately when an instruction with characteristics of locality appears, taking full advantage of the locality. Two branch predictor-inspired policies are developed. Experiment results indicate these policies bring up to 60% of the maximum possible benefit for task migration.

A Template Library to Integrate Thread Scheduling and Locality Management for NUMA Multiprocessors

Zoltan Majo and Thomas R. Gross, ETH Zurich

Zoltan Majo presented a template library that can explicitly divide data and threads among different processors. In a NUMA system, reducing remote memory access is critical to improving performance. This can be achieved by data partitioning so that each processor can access locally. Also, thread execution should be assigned appropriately to guarantee that data required by a thread is local. Although there are many researchers who focus on NUMA systems, few of them have investigated the problem of data locality in NUMA systems. In this presentation, Majo first showed a case study of ferret, a program from the PARSEC benchmark suite, and then discussed a template library for explicitly partitioning data and thread execution.

In Majo's case study, 16 threads of ferret run on a two-processor eight-core system. The memory access behavior is recorded by data address profiling, and only memory access to heap is considered. According to the percentage of accesses from the main processor, all memory accesses are divided into six categories. Majo explained the characteristics of these six types and also analyzed the reason for interprocessor data sharing in ferret.

After the ferret case study, Majo introduced a template library that enables programmers to explicitly allocate data and thread execution. This library allows programmers to pre-define data distribution and thread-scheduling primitives. Experiment results show that remote memory accesses are reduced from 42% to 10%, and the overall performance is improved by 3%.

Panel: Parallel Programming in the Real World

Summarized by Mohamed Mohamedin (mohamedin@vt.edu)

Moderator: Luis Ceze, University of Washington

Panelists: Andrew Brownsword, Intel; Niall Dalton; Goetz Graefe, HP Labs; Russell Williams, Adobe

Parallelism in the "Real" World

Andrew Brownsword

Andrew started by defining what real software is: large, written by many people, expressed by program models, having many levels of abstraction; and it should be fast, robust, and maintainable. He used games as an example for real software. Games are medium to large, may have a large toolchain and online infrastructure, have many diverse subsystems running together (e.g., graphics, environment, audio, animation, game logic, AI, physics), are soft real-time, frame oriented, sequential dependency, and have many target devices (e.g., from phones to servers, different CPUs, multicore, GPUs). Moreover, games have short development cycles, changing rapidly, substantial volume, and usually are ported between diverse platforms.

Andrew then talked about HPC (high performance computing). HPC is characterized by wide varying code bases and domains (e.g., C/C++, Fortran, MPI, OpenMP), few kernels, big data, and different target platforms. The HPC development process is ongoing, and it should be correct and portable.

Programming models are very important. They must have the following properties: integration (i.e., with other models, runtimes, etc.), portability between different hardware and OSes, composability, concurrent execution on multiple levels (e.g., vectorization, parallel cores, distributed nodes), data organization and access patterns (i.e., FLOPS are cheap but memory bandwidth is not), and specialization.

Stock Market Environment

Niall Dalton

The stock market handles billion of messages every day. It must be able to handle large amounts of data in memory and respond to messages in real time without faults. In this environment, they use FPGAs, GPUs, manycores, and SOCs. We shouldn't fight the last war; we started with the clock speed war, followed by the current core war. Instead, we now have the efficiency war, which concentrates on pure throughput.

In order to support the stock market's special environment, they used custom solutions based on special FPGA hardware, custom switches, SSD, and a special programming language that looks like a mix between SQL and Matlab. Using this custom language, one can easily run parallel jobs on huge amounts of data.

They are facing problems that are still open. One problem is machines are already beyond our ability to program productively with high performance. Another problem is it is getting harder to observe, understand, debug, and tune our programs/machines.

Hot Topics in Parallelism in Data Management

Goetz Graefe

Historically, there's been concurrency among independent database transactions (single-threaded) since the 1960s, and parallel query processing since the 1980s. Database transactions follow the ACID (atomicity, consistency, isolation, and durability) model and can be divided into two types: user and system transactions. User transactions are those visible to the user in the form of queries and updates. They lock the database to commit, and if the commit fails, the changes are rolled back from recovery logs. System transactions handle the internal representation of the database and how the data structures are accessed.

The current trend is to focus on scalability. Techniques like MapReduce (e.g., Hadoop), data mining, and business intelligence are used to handle large databases. Also, new implementation techniques—such as transactional memory, a new synchronization abstraction based on database transactions—are required to handle low-level synchronization.

Parallelism in Adobe Photoshop

Russell Williams

Photoshop is a huge cross-platform program based on a single thread. Parallel computations were designed as a general framework in the mid-90s and do not scale well after four cores. It trades off throughput for latency and maintains a large pool of unused threads.

The structure of the problems facing parallel desktop software is whether we do asynchronous actions or parallel computations that need synchronization. What are the sources of parallelism? Are we using events and views or are we controlled by Amdahl's Law? FLOPS now are cheap but the bandwidth is limited; we can have 80-core chips but the software cannot use them. We have a heterogeneous environment (e.g., different cache designs, different languages), and the hardware is rapidly changing (e.g., SSE, AVX, AVX2). All this makes developing software over the next five or 10 years very difficult as hardware and software tools keep changing.

One major problem in parallel computations is that we have a large number of FLOPS that are not utilized. For example, in an eight-core machine with AMD 6950, we have two GFLOPS in the GPU, 0.5 GFLOPS in CPU vector unit, and a very small fraction of GFLOPS in multithread and scalar core. On the other hand, 99% of the code is written for the scalar core. Moreover, most programmers' expertise is in this area.

Discussion

Since the bandwidth is the major problem in parallelism, can we formulate the problem so that we have a framework where we can decide on the data flow and where the data is placed in the memory hierarchy? The problem is most of the programs are written in C/C++. They follow the same order of fields in structures. Even if we had a magic compiler that could reorganize the data, optimization would be very difficult. For example, optimizing just two functions took a month. Maybe the solution is to have a domain-specific language like the one used in the stock market. Someone else mentioned that debugging in some new languages is very difficult: for example, OpenCL. Another person pointed out that one of the problems with C/C++ is lack of reflection so that all variable names get lost in compilation, making optimization and data rearrangement difficult.

Someone else mentioned that most of the talks have been about performance. Does this mean that the current tools are enough? We still need good tools for debugging and for handling bugs dynamically. Another panelist commented that we need to choose one technology and focus on it. This is why OpenCL is not common and has a small code base. Another person wondered why there is so much concentration on computing power only. What about network latency? A panelist replied that we consider latency too. For example, our private network between two main cities is very fast and has low latency.

Amazing Applications

Summarized by Brandon Myers (bdmyers@cs.washington.edu)

Retrofitted Parallelism Considered Grossly Sub-Optimal

Paul E. McKenney, Linux Technology Center, IBM Beaverton

Paul McKenney introduced his talk by referring to the human fascination with mazes, showing an example of an ancient maze dug into the earth. "Not much has changed since then except now our mazes are stored in silicon instead of silicon dioxide." A straightforward sequential maze-solving algorithm consists of storing branch points and revisiting them until the endpoint is found (SEQ). A common approach to parallelizing this solver is to have multiple threads, all adding and removing branch points to a work queue (PWQ). This approach can give up to linear speedup. The weak point of PWQ is that at most one thread is ever making progress along the solution path at a given time.

Another approach is to trace from both the start and the end with two threads. This algorithm is called PART, since it is analogous to the parallel programming approach of partitioning the problem into two. Run on the random test mazes, this version solves mazes up to 40x faster than SEQ. This superlinear speedup makes the problem appear "humiliatingly parallel": adding threads reduces the total work. Why is this? For PART, when threads get in each other's way, rather than causing contention, it helps because as soon as a thread sees a branch point already covered by the other thread, the solution has been found. The superlinear speedup is statistical; a random maze is likely to benefit from PART, but in the worst case, all maze cells may have to be touched (e.g., for the trivial maze with no branches). Given that PART gives superlinear speedup, Paul found that a co-routine implementation gets much of the speedup of parallel PART, although having two threads provides some advantage for larger mazes. He also compared sequential compiler optimizations to parallel speedups.

Paul's high-level takeaways included parallelizing a sequential algorithm is not always the best approach; parallelism should be treated as first class; and be aware of parallel results that are vulnerable to better sequential optimizations. To the question, do all humiliatingly parallel problems have good co-routine implementations, Paul thinks perhaps some might not: for example, if context switch overhead outweighs the computation.

Parakeet: A Just-In-Time Parallel Accelerator for Python

Alex Rubinsteyn, Eric Hielscher, Nathaniel Weinman, and Dennis Shasha, New York University

Alex Rubinsteyn presented work on JIT parallel acceleration of Python code using a GPU. Dynamically typed, interpreted languages like Python can enable high productivity but have poor efficiency compared to low-level C++ or CUDA code. Python application programmers with numerical workloads can get better performance by using the NumPy library, which uses precompiled functions like sum. The deficiency is that most computations using NumPy are still locked to one core, except for some functions that use Basic Linear Algebra Subprograms (BLAS). We want to do better, while staying completely within the Python programming language. The Parakeet library exposes a variety of adverbs, higher order functions that can be run safely in parallel if provided pure functions. Some of the adverbs that Parakeet provides are map, reduce, scan, and allpairs, which consists of nested maps.

Parakeet supports a very limited subset of Python; for example, functions cannot use any structures other than scalars and NumPy arrays. What is left makes Parakeet an "array calculator," but if your Python program has a bottleneck then writing that part in a subset might be tolerable. Providing the parallel adverbs already allows many data parallel computations to be expressed.

Parakeet calls can be parallelized currently as either a GPU kernel or using a work queue for multicore CPUs. On each call to Parakeet, the runtime must reanalyze the intermediate representation to infer the shape of data for GPU preallocation and decide which level of the computation to parallelize. Alex has found that single-core BLAS operations can still perform far better than Parakeet. The lesson is that data layout and cache-locality are critical. Future work will improve locality, use a more portable GPU back-end, and use a better cost model for accurately deciding what work to assign to the GPU or CPU.

Discussion centered on programmability tradeoffs. Luis Ceze commented that accelerator or parallelization approaches like Parakeet can sacrifice programmability and was curious whether one can support a subset of Python that is interesting and useful enough. Alex replied that it is too early to tell whether restrictions that Parakeet code sections require will deter adoption. But he argued that from his experience, many Python programmers who need more performance would prefer to stay within a .py file (as opposed to, e.g., having a hybrid C and Python program) even if they have to write part of the application in a restricted way. David Reed asked whether the subset of Python can be expanded. Alex answered that as soon as you move beyond arrays of data or allow dynamic features, parallel implementation gets far more complicated.

Test and Be Safe

Summarized by Mohamed Mohamedin (mohamedin@vt.edu)

Concurrency Attacks

Junfeng Yang, Ang Cui, Sal Stolfo, and Simha Sethumadhavan, Columbia University

Junfeng Yang said that concurrent programs have many hidden bugs, which can be called concurrency attacks. These attacks are difficult to exploit. The authors found 46 exploitable concurrency errors in famous programs. For example, Moonlight has such an attack when code copies an array. The code has no problems in sequential execution. A second example is that Internet Explorer can have a data race due to an appendChild bug. A third example is that in iOS there is a physical proximity attack that can allow access to the phone without entering the passcode. Their study shows that such errors are pervasive.

There are some factors that affect exploitability of these attacks. One of them is the vulnerable window size. In physical proximity attacks, the window size is in seconds. It is in milliseconds in file-system-related bugs and microseconds in memory-related bugs. An attacker can enlarge the window size (e.g., increase the array size in the Moonlight bug). Also, attackers can catch the small window size by programmatically retrying (e.g., the IE bug).

Concurrency attacks cannot be prevented by current defense techniques, which are designed for sequential execution. For example, metadata tracking, software checks, and anomaly detection are all weakened while other defenses like hardware checks are not affected.

Someone wondered whether removing data race prevents SQL attacks, Junfeng Yang answered no, since this data race is not in the original program, it is injected by the attacker. There was also a comment that other types of concurrency attacks are not mentioned in this study such as the one that allowed finding a password by monitoring the cache.

CONCURRIT: Testing Concurrent Programs with Programmable State-Space Exploration

Jacob Burnim, Tayfun Elmas, George Necula, and Koushik Sen, University of California, Berkeley

Tayfun Elmas began by saying that it is difficult to write a unit test for concurrent programs. In sequential ones, it was enough to fix the input and use assertions. In concurrent ones, they need to fix the schedule as well as the input. Stress testing gives no guarantees, and model checking is not applicable to large programs as it requires testing all schedules; thus, they need to control the program schedule.

The current technique of using "sleeps" is considered ad hoc and not formal. Concurrit provides a DSL for writing concurrent tests. The software under test (SUT) is instrumented so that it blocks after each event and waits for the continue signal. A test case consists of events (conditions), and the response to these events determines the number of schedules in the test case. Choosing the right conditions leads to fewer schedules, so they can detect the bug in a short time. Concurrit is available online at http://code.google.com/p/concurrit/.

Someone asked whether a deadlock can occur in Concurrit. Elmas answered yes; they are handled using timeouts and backtracking.

Understanding the Interleaving-Space Overlap Across Inputs and Software Versions

Dongdong Deng, Wei Zhang, Borui Wang, Peisen Zhao, and Shan Lu, University of Wisconsin, Madison

Testing a program using data-race detections results in a 10–200x slowing of program execution. The problem is that when a new version is tested, a lot of tests are redundant. Wei Zhang described this work as finding redundancy across inputs of the test cases without doing a slow interleaving space analysis.

The first step is to write concurrent function pairs (CFP), then design the testing plan. The final step is running the CFP-based race detection. This system is tested and did not miss any failure-inducing data races. Results shows that this new method reduced the number of test cases that produce the same error significantly. In some cases, they reached only one test case, which is the best solution.

Currently this work is limited to data races only. Supporting detection of redundancy in other types of bugs is planned future work.