2nd USENIX Workshop on Hot Topics in Parallelism (HotPar '10)

June 14–15, 2010
Berkeley, CA

JUNE 14, 8:30 A.M.–10:00 A.M.

Summarized by James C. Jenista (jjenista@uci.edu)

- ***Towards Parallelizing the Layout Engine of Firefox***
  Carmen Badea, University of California, Irvine; Mohammad R. Haghighat, Intel Corporation; Alex Nicolau and Alexander V. Veidenbaum, University of California, Irvine

Multicore is ubiquitous and the browser is becoming a thin client to run a wider range of applications. Carmen Badea argued, therefore, that it is worthwhile to explore the parallelism in browsers.

Badea explained that Firefox was chosen because it is open source and has the second highest browser market share. They profiled Firefox and discovered that 40% of the test execution time was spent in the layout engine and 32% of that time is devoted to CSS rule matching. This led to a parallelization effort of the CSS rule matching subsystem. After giving a brief background for CSS, Badea explained that CSS rule matching executes when a user loads a new page or a page is interactively updated. The Mozilla Firefox page load tests and Zimbra Collaboration Suite (ZCS) were employed as benchmarks to profile the CSS rule-matching system; descendant selector rules were executed most often, and the vast majority resulted in a non-match.

They decided to parallelize the common descendant rule case of non-match by executing rules for batches of ancestors of an element concurrently. When there is a rule match, some of the work is speculative and therefore discarded, although Badea argued that the profiling data implies this case is infrequent. Dan Grossman asked how many rules are being matched in the parallel implementation, and Badea answered that only one rule is matched at a time. She explained that the code base is factored this way, although future work could explore a parallel implementation that matched many rules at once.

The parallel CSS rule matcher was tested in seven configurations for ZCS and in 12 configurations for the Mozilla test pages. Badea noted that more than two threads did not perform well and hypothesized that future Web pages with richer CSS may benefit from more than two threads. For Mozilla pages, the end user's perceived speedup was as high as 1.8 times, and for ZCS as high as 1.6 times. They attribute the better speedups for the Mozilla page load tests to more complexity in layouts, as well as to the fact that ZCS is a more JavaScript-oriented benchmark suite.

Badea was asked if she believed there will be fewer improvements for such a parallel CSS rule matcher as Web pages have more and more JavaScript. She answered that more JavaScript doesn't exclude more complex layouts. Can an early match result in a longer execution time than the single-threaded version? It's possible, but the profiling data suggests this is a rare occurrence. What behaviors caused the worst speedups? Badea explained that Web pages with few ancestor elements did not trigger the parallel rule matcher, but suffered from added preprocessing.

- ***Opportunities and Challenges of Parallelizing Speech Recognition***
  Jike Chong, University of California, Berkeley; Gerald Friedland, Adam Janin, Nelson Morgan, and Chris Oei, International Computer Science Institute

Adam Janin said that the goal of their work is not for the sake of parallelism specifically, but, rather, to improve speech recognition accuracy, throughput, and latency. Janin then offered a scenario to drive his presentation; he had recorded the speech of a meeting with an iPhone on the table. The systems they developed should process the audio and allow browsing and retrieval of useful information such as querying who was speaking at a given time, finding audio segments by words spoken, and finding segments by speaker. Janin broke down the system and made a clear distinction between speech recognition that extracts words and diarization that identifies the speaker.

Then Janin built an argument for developing a parallel software implementation. Current technologies scale easily along any resource axis; still, state-of-the-art systems are 100 times slower than real time to achieve the best results. Specialized hardware has gotten mixed improvements, so general parallel software may be the answer.

Perceptual models of the inner ear, Janin explained, are used to compute features of audio. The combination of features usually improves results for noisy conditions, so current systems typically select two to four cochlear representation variants. Janin asked, when we have more resources, why not add many more representations? He explained that the representations are filters fed to a neural net, which prompted a question: is the system similar to deep neural nets? Janin answered yes, he would call it deep learning, but with no unsupervised step. He then highlighted that the many streams and dense linear algebra required all have an obvious parallel structure.

Their experiments included both a 4-stream and a 28-stream configuration. Janin indicated that the 4-stream setup improved accuracy by 13.3% on a Mandarin conversational task, and the 28 streams improved accuracy by 47% on a read digits tasks (e.g., phone numbers, zip codes). When asked if the system is commercially viable, Janin answered that the noisy number input audio is an artificial test, and current systems can do well for reading numbers over the phone under normal noise conditions. Another questioner asked whether there is a diminishing return for adding streams. Janin responded that they don't know, but he certainly believes it. The data supports it, although he

said the brain is thought to be processing hundreds of millions of audio interpretations at once.

Janin then moved from their improvement of accuracy to the improvement of throughput. They pipelined the speech recognition system and improved the throughput of the inference engine, which looks up the closest utterance from a language. Janin described how the machine-learning model generates a complex, static graph of state transitions to implement the inference. The online system, he explained, does a time-synchronous beam search over the graph, only keeping the best hypotheses. When asked how big the graph is, Janin answered that there are one million states, four million arcs, and thirty-two bytes for each node. They reported an 11-fold speedup overall—an 18-fold speedup for the compute-intensive transitions, and a 4-fold speedup for communication-intensive hypothesis merging.

The next segment of Janin's talk covered how they improved latency and accuracy for online diarization. This might be useful, Janin explained, to identify who is speaking in real time during a distributed meeting. An attendee asked if training data is needed. Janin responded that none is needed for the speakers or even the language. Their strategy is to begin the offline diarization as soon as the meeting starts and hand off models for each speaker to the online system as they improve over the course of the meeting. Janin reported the error rate drops about 7% by parallelizing this implementation over eight cores.

In response to several requests to characterize the challenges of developing the parallel software, Janin replied that designing the parallel algorithm was more challenging than the implementation. New parallel tools could certainly help, especially any that might bring in new programmers. David Padua asked if there is a way to measure the progress in the field of speech recognition. Janin gave details of the US government's annual challenge. Janin's analysis was that the accuracy of systems entered has slowly improved to about 50% word error rates, which he said is quite good for many applications, but that the major progress in the field has been to accomplish previously hard tasks much more easily.

JUNE 14, 10:30 A.M.–12:30 P.M.

Summarized by Chris Gregg (chg5w@virginia.edu)

- ***A Balanced Programming Model for Emerging Heterogeneous Multicore Systems***
  Wei Liu, Brian Lewis, Xiaocheng Zhou, Hu Chen, Ying Gao, Shoumeng Yan, Sai Luo, and Bratin Saha, Intel Corporation

Brian Lewis talked about how computer architecture is becoming more heterogeneous and how to improve programming models for such systems. More and more programmable accelerators are being designed into computer systems, and this talk focused on them. GPUs are either discrete or integrated on-die with CPUs, in which case they share computational resources. Low-level languages that exist today (OpenCL, CUDA) focus on coarse-grained offloading of parallel computation, but do not fully take advantage of CPU capabilities. The authors want to improve programmer productivity and extend the range of applications that can be easily programmed.

David Padua asked, "Is the limitation for fine-grained processing a factor of language, or of hardware?" Lewis answered that both were relevant, to an extent. It is low-level, which does not lead to high-level breaking up of tasks. There was another question about why parallel languages weren't yet meeting our needs; Lewis answered that it is mainly because they are still relatively low-level. They want a balanced programming model, to enable fine-grained computation using all cores, with better support for task and data parallelism, load balancing, and dynamic reconfiguring.

Lewis talked about the importance of shared virtual memory and the need for lightweight atomics and locks, which will allow better coordination between the CPU and GPU. The discrete Larrabee implementation has shared memory that supports release consistency and ownership rights, which allows the CPU and GPU to both work on the same data. There is an OS on both sides, leading to VM page protection, which helps with consistency. The shared memory CPU-integrated graphics has a device driver, and there isn't an OS to handle page faults. It doesn't detect updates using page faults, but it exploits shared physical memory, meaning there is no data copying.

Nicholas Matsakis asked, "Is there a model for how the data should be shared?" Lewis said that the keyword "shared" marks shared data. Keywords, used for offloading functions, are elaborated on in the paper. Timothy Roscoe asked, "Have you looked at what would happen if you ran multiple applications across this system?" Lewis answered that they did not look at that for this paper, but that is the end goal of the work.

- ***Collaborative Threads: Exposing and Leveraging Dynamic Thread State for Efficient Computation***
  Kaushik Ravichandran, Romain Cledat, and Santosh Pande, Georgia Institute of Technology

Romain Cledat started by discussing parallelism in general and how it can be improved. Parallelism today relies on threads, which is splitting up data or tasks. Current models include TBB and CnC, which leads to a natural parallelism. However, threads still use locks and barriers and transactional memories. They share data through shared memory, but do not have knowledge about their "role" in the computation nor the overall state of the computation. Current models break up a computation, and the distribution of work is done just in time. The state of the computation is not taken into consideration. The threads work independently and do not have higher-level semantic knowledge. Performance of HPC problems has dependencies that are greater than simply how the work is split up.

Cledat then discussed useful semantic state, determined by the programmer, to influence scheduling. Byn Choi asked, "Isn't scheduling the threads the role of the task scheduler, and are you trying to make the threads do this in a distributed manner?" Cledat said they are trying to do more than scheduling and were trying to use the meta-information for more than just this problem.

Cledat turned the talk over to Kaushik Ravichandran, who discussed the system the authors created, specifically the semantic state taxonomy. Similar sub-problems are clustered together in a tree structure. The sub-problems are hierarchical, incremental, and approximate. This results in good lookup time, without having to build from scratch. With this information, they can re-use results, orient a computation, prioritize sub-problems, and select cores appropriately. Sean Halle asked, "Is the compiler doing this, or the app programmer?" Ravichandran answered that the programmer designates certain information and the run-time uses it.

Ravichandran provided an example of a sum of subsets, showing that a large of amount of redundancy can be exploited. The programmer can more aggressively parallelize this problem by specifying a similarity metric, which the system will use to make the best use of previously computed values. For the second example, K-Means, objects can share data between localized points. The speedup comes from making fewer comparisons than the original algorithm, sharing results from the closest neighbors.

- ***Structured Parallel Programming with Deterministic Patterns***
  Michael D. McCool, Intel

McCool discussed how people parallelize applications and the structures they use. In particular, he described a total of 16 different fundamental parallel programming patterns. A *parallel pattern* is a commonly occurring combination of task distribution and data access. Many programming models support a small number of patterns or low-level hardware mechanisms. However, a small number of patterns can support a wide range of applications, deterministically. A system that directly supports the deterministic patterns on a lot of different hardware architectures can lead to higher maintainability, and application-orientated patterns can lead to higher productivity.

Sean Halle asked, "Should patterns not have hardware details?" McCool answered, no, he would rather find more abstract patterns, and specifically functional programming patterns. There are structured programming patterns for serial computation, and we can add a number of parallel patterns to this list for a number of different, fundamental patterns.

Sean Halle asked, "Do you want your application talking to the runtime?" and McCool replied that yes, although you don't want to over-constrain the runtime. You do, however, want communication between the two. He continued his talk by discussing partitioning, which is very important; you're breaking an input collection into a collection of collections. This is useful for divide-and-conquer algorithms. There is also the issue of boundary conditions. Another pattern is stenciling, which applies a function to all neighborhoods of an array. There are also fused patterns that can be useful in specific conditions. Examples include: gather = map + random read; scatter = map + random write. Scatter is tricky, because you need to watch out for race conditions. It would be nice to find a deterministic scatter, and the best solution is "priority scatter," which prioritizes the elements as they would have happened in a scalar scatter.

McCool finished with "the bottom line," trying to create a taxonomy of good practices for parallel programming. Are these the right patterns? Is there a smaller list of primitive patterns? How important are nondeterministic patterns? Sarita Adve asked about determinism and isolation, and McCool answered that the merge-scatter pattern came closest to matching.

JUNE 14, 12:30 P.M.–2:00 P.M.

Lunches on both days included tables labeled with questions for discussion. You can find the results of these discussions and some comments at http://www.usenix.org/events/hotpar10/tech/techLunches.html. (I found the results fascinating and interesting in themselves.—The Editor)

JUNE 14, 2:00 P.M.–4:00 P.M.

Summarized by James C. Jenista (jjenista@uci.edu)

- ***Separating Functional and Parallel Correctness using Nondeterministic Sequential Specifications***
  Jacob Burnim, George Necula, and Koushik Sen, University of California, Berkeley

Jacob Burnim identified nondeterministic interleavings as a major difficulty when reasoning about the functional correctness of a parallel program. He proposed that a programmer-generated nondeterministic sequential artifact could decompose the effort into the questions of parallelism correctness and functional correctness. The key, Burnim explained, is that the programmer should annotate intended nondeterminism and then a system can check that the parallelization adds no more nondeterminism.

As an example, Burnim introduced a branch-and-bound code and asked the attendees to consider the sequentially expressed code as a parallel version by adding a few parallel constructs; is the parallelization correct? Burnim offered an interleaving that shows that the parallel answer may be different but correct. Burnim hypothesized that a specification in between the sequential and parallel codes is needed to express the allowed nondeterminism and then provide a framework for proving the correctness of the parallelization.

Their artifact is a nondeterministic sequential (NDSEQ) expression of the code. Burnim introduced the nondeter-

ministic for loop as an element of the NDSEQ which runs one iteration at a time but in any order. Burnim pointed out that there are still interleavings to avoid some prunings that the parallel version can express but the NDSEQ cannot. As there is intended nondeterminism in the example, Burnim introduced the use of if(*) to instruct the NDSEQ to choose either branch. Burnim claimed the modified NDSEQ expressed the intended nondeterminism in the parallel version, and that the NDSEQ could generate a given parallel interleaving. A question was raised about whether the parallel algorithm was suboptimal, which Burnim conceded, but he stated that it is reasonable and apt for the illustration of their work.

Once the NDSEQ is provided, Burnim continued, the parallel correctness and functional correctness can be proved separately. He interjected an argument that the correctness of the parallel version is undecidable and the correctness of the NDSEQ is decidable, offering another justification for the effort of creating the NDSEQ. Then Burnim demonstrated the correctness of the parallelism with a proof by reduction, consisting of the rearrangement of parallel interleavings matched against the NDSEQ. Burnim was asked if the proof works for nested loops. He said that the correctness of the inner loop can be proved, then replaced with a sequential version to prove correctness of the outer loop.

Their future work will include automating the proof for real benchmarks. Burnim suggested that their approach might be applied to other model checking techniques. He also suggested that instead of a static system, their work might be integrated in a debugger to consider the correctness of a parallel trace, where a bug might be classified in relation to the parallelism or the functional correctness.

An attendee asked how to detect when the NDSEQ is incorrect. Burnim answered that there are two cases: when the NDSEQ is too strict, the situation is manageable, as the system could report parallel interleavings that the NDSEQ cannot express to aid NDSEQ improvement; when the NDSEQ is too weak, Burnim conceded that it becomes a difficult problem. Several people asked Burnim about the possibility of language solutions to avoid needing a correctness checker. Burnim answered that when you get correctness for free, language solutions are good, but some computations, such as types with a lot of guarantees, are hard to express without sufficient nondeterminism.

- ***Synchronization via Scheduling: Managing Shared State in Video Games***
  Micah J Best, Shane Mottishaw, Craig Mustard, Mark Roth, and Alexandra Fedorova, Simon Fraser University, Canada; Andrew Brownsword, Electronic Arts Blackbox, Canada

Micah Best introduced their work as a fruitful technique for synchronizing threads via scheduling in the video game domain, a domain in which performance and responsiveness are high priorities. Though it was not the subject of his talk, Best covered the Cascade project, which expresses a video game engine as a dataflow task graph. Their work integrates with Cascade, he explained, and attempts to ease the burden of managing task-shared state off the developer through static analysis and new synchronization techniques at runtime.

Best described how static analysis of the Cascade mark-ups identifies constraints between tasks. The constraints are potential conflicts, such as access to elements of a collection, and their work uses a runtime strategy to determine the actual constraints. Best stated that the scheduler uses task-constraint profiles to synchronize access to shared data.

Best moved the discussion to a method of expressing constraints in binary. References and members, he said, are simply expressed. He continued with the expression for arrays which occur frequently in video game kernels and require some analysis of indices. The hardest cases are forms of indirection and will be addressed in their future work.

The constraints identified by static analysis are passed through Bloom filters to produce a fixed-length bit string. The bit string is a constraint signature for the task; Best added that signatures are cheap to calculate and compare. A task may run when its signature has no conflict with running tasks, although signature comparisons may produce false positives but will never show a false negative. In response to a question about user control over the signatures, Best responded that users may tune the construction parameters through Cascade.

Best characterized their scheduling algorithm as generational. Tasks are batched by using logical-OR on their signatures until no more tasks may be added without conflict. A batch forms a generation and is sent to a core while the next generation is batched.

They tested their work by adding Cascade mark-up to Cal3D, a library for animating character models where separate animations may be blended and applied to the same model. When the application of multiple animations have a state conflict, the system synchronizes access; otherwise animations may be applied concurrently. Best then presented the experimental setup; a workload of four models with eight animations was executed on a two-processor Xeon totaling eight cores.

Their results were compared to a natural implementation as a baseline, which Best defined as one written by a non-expert, competent programmer. The baseline implementation applies animations in a straightforward way without requiring synchronization. Best displayed an activity graph from Cascade that showed banding effects in core usage because there was not enough work while waiting for the next animation. With signatures and then a partitioning strategy they obtained better core utilization. Best highlighted an important result by presenting an expert-tuned version of the benchmark that had the highest utilization. He concluded that they had pursued parallelism too aggressively; a method for finding the right amount of parallelism for given overheads is future work.

Someone from Toshiba asked how they could encourage adoption. Best answered that adoption is always a concern for new parallel languages. One approach is to convince programmers the benefits of the new language are undeniable and always be sure the language is addressing the true problems facing the programmer. Nicholas Matsakis asked how much work Cal3D was to port. Best replied that the work was completed in a few weeks but noted that porting the Cube 3D code was much more difficult. He attributed this to the well-written source for Cal3D as opposed to messy source for Cube 3D and concluded that bad code is hard to parallelize.

- ***Get the Parallelism out of My Cloud***
  Karthikeyan Sankaralingam and Remzi H. Arpaci-Dusseau, University of Wisconsin—Madison

Karthikeyan Sankaralingam asked whether the current degree of focus on parallelism and multicore is out of proportion to the number of applications for the research. Implementing parallel software is complex, Sankaralingam said, and by asking if real developers or users even want it he stirred up a hornets' nest.

Sankaralingam painted a future computing environment in which notebooks and smartphones offload computation to the cloud, and the average programmer can easily deploy and maintain software in the cloud. He argued that a small number of experts can implement the lower layers of the cloud for multicore architectures, while the average programmer or user device sticks with a few-core model.

Their work addressed three myths that Sankaralingam hypothesized are steering research away from improving the cloud environment and toward an overemphasis on multicore and parallelism. The first myth Sankaralingam covered was that hardware drives software. He argued that programmers historically spent significant software effort to achieve efficiency with hardware, but the major hardware problems are now solved. Now, he continued, programmers must be productive and demand high-level languages to express programs with as little code as possible, and therefore software is currently either decoupled from or even driving hardware.

Sankaralingam moved on to the second myth: multicore will be everywhere. He presented a graph describing the relation of performance to energy and explained that technology scales the curve, but by only so much, and ultimately the number of cores on a handheld device is limited. Sankaralingam conjectured that the limit will be about 10 cores. An attendee asked what he meant by a core; Sankaralingam said he meant a programmable processor. He concluded his discussion of this myth by pointing out that the mobile device may not need multicore, because from its perspective it gets free performance from the cloud without paying energy.

The third myth Sankaralingam identified was that everyone should become a parallel programmer. Sankaralingam called parallel programming a great challenge that may even disrupt the curriculum and suggested it should be left to the experts. The average cloud application parallelizes over many clients in the cloud without being a parallel program, he said.

Sankaralingam summarized their work as an argument to rethink the role of parallelism and then opened for questions by taking off his jacket, revealing a bull's-eye emblazoned t-shirt. Krste Asanović stated that productivity will always be important, and Sankaralingam agreed but used Jango as an example of programmers never even seeing the underlying SQL base. Someone from Qualcomm disagreed that cloud computing will become dominant, because distance to the tower doesn't follow Moore's Law, but devices are following it. Sankaralingam agreed that latency is a hard problem in cloud computing, but offered an anecdote. Sankaralingam had mounted a remote file system while traveling to the workshop, with virtually no impact on his environment; already, he said, the latencies are not so apparent to the end user. Sean Halle began by saying that Sankaralingam was very brave, and Sankaralingam replied that his advisor, Remzi Arpaci-Dusseau, is responsible for the things you disagree with. Halle pointed out that there are 200,000 iPhone applications and asked if Sankaralingam believed the iPhone successor will be single-core. Sankaralingam answered no, but continued by claiming that a mobile device will never have 100 cores for the average programmer to deal with. Sarita Adve asked who the PC members were who accepted this paper, as she wanted to talk with them later.

JUNE 14, 5:00 P.M.–8:00 P.M.: POSTER SESSION

Posters below summarized by Romain Cledat (romain@gatech.edu)

The poster session included all the talks in the program, as well as the papers reported here.

- ***A Principled Kernel Testbed for Hardware/Software Co-Design Research***
  Alex Kaiser, Samuel Williams, Kamesh Madduri, Khaled Ibrahim, David Bailey, James Demmel, and Erich Strohmaier, Lawrence Berkeley National Laboratory

In this work, the authors developed high-level language implementations of key kernels in HPC. They then implemented each kernel in C. The kernels cover the seven Dwarfs presented in the Berkeley vision. A tech report as well as the full code in C will be released soon. Note that all implementations are sequential. Contact: ADKaiser@lbl.gov.

- ***Contention-Aware Scheduling of Parallel Code for Heterogeneous Systems***
  Chris Gregg, Jeff S. Brantley, and Kim Hazelwood, University of Virginia

This work looks at how best to choose where a program needs to run: on the GPU or on the CPU. The assumption is that most kernels will prefer the GPU but it depends on

whether the GPU is busy, the input size, the runtime of the baseline, the historical runtimes for the program, etc. Contact: chg5w@virginia.edu.

- ***Capturing and Composing Parallel Patterns with Intel CnC***
  Ryan Newton, Frank Schlimbach, Mark Hampton, and Kathleen Knobe, Intel

This work extends the CnC model by introducing modules which encompass an entire graph as a single step. This allows better reusability of code and modular building. CnC also introduces more schedulers for the tuning experts. The TBB scheduler is still the base scheduler, but there are now schedulers to specify task priorities, ordering constraints, and locality. Contact: ryan.r.newton@intel.com.

- ***General-Purpose vs. GPU: Comparison of Many-Cores on Irregular Workloads***
  George Caragea, Fuat Keceli, Alexandros Tzannes, and Uzi Vishkin, University of Maryland, College Park

This work presents a PRAM-on-chip vision with a full vertical integration from the PRAM model to the hardware implementation. XMT is the PRAM abstraction and XMTC is the C-like language built on top of it. The PRAM model provides speedup in many cases, as well as ease of programming. Furthermore, there is no need to reason about race conditions. This model has been tried in classes and people get it very quickly. Contact: {gcaragea,keceli,tzannes, vishkin}@umd.edu.

- ***Leveraging Semantics Attached to Function Calls to Isolate Applications from Hardware***
  Sean Halle, INRIA Saclay and University of California, Santa Cruz; Albert Cohen, INRIA Saclay

The need for continuity with past systems in parallel programming makes function calls very attractive (similar to OpenGL). Indeed, big changes are expensive and take time, and people feel comfortable with the way they were doing things before. After the code has been written to integrate the parallel function calls, a specializer can produce different implementations for each call depending on the platform. The code is therefore isolated from the platform. Furthermore, this specialization step happens after the main development cycle, which means that there is more time to do it right. Another important aspect of the model is the use of program virtual time to easily detect scheduling errors. The final aspect of the model is the use of interfaces to implement paradigms such as "divide work." The application implements an interface "how to divide" which the runtime can call with the number of chunks to produce, depending on the target platform. Contact: seanhalle@yahoo.com.

- ***Enabling Legacy Applications on Heterogeneous Platforms***
  Michela Becchi, Srihari Cadambi, and Srimat Chakradhar, NEC Laboratories America

The goal of this work is to enable the re-targeting of legacy applications to heterogeneous systems. The system uses libraries to catch certain system calls, and each platform can have its own library which implements the calls differently depending on the platform. Contact: mbecchi@nec-labs.com.

- ***OpenMP for Next Generation Heterogeneous Clusters***
  Jens Breitbart, Universität Kassel

This work is an extension of OpenMP. It works on shared memory systems and adds PGAS-like semantics for distributed memory systems. In that case, the runtime will seek to over-saturate the system to hide latencies. Annotations are done just as in OpenMP. Contact: jbreitbart@uni-kassel.de.

- ***Energy-Performance Trade-off Analysis of Parallel Algorithms***
  Vijay Anand Korthikanti and Gul Agha, University of Illinois at Urbana-Champaign

Energy is becoming a big issue: as performance increases, energy increases quadratically. For embarrassingly parallel applications, increasing the number of cores is good, as it results in better time and a quadratic decrease in energy. The problem, however, lies in the energy required to communicate. There is a sweet spot that optimally trades off communication energy and core energy. Two metrics are introduced: energy scalability under iso-performance and energy bounded scalability. The goal of this work is to determine the optimal number of cores based on the input size. Contact: vkortho2@illinois.edu.

- ***Prospector: A Dynamic Data-Dependence Profiler to Help Parallel Programming***
  Minjang Kim and Hyesoon Kim, Georgia Institute of Technology; Chi-Keung Luk, Intel Corporation

This work introduces Prospector, a profiling approach to dynamically determine data-dependencies. This greatly improves auto-parallelization. The main contribution of this work is the implementation of efficient compression of the profiling data. This produces much better results than Intel Parallel Advisor, for example. Contact: minjang@gatech.edu.

- ***Bridging the Parallelization Gap: Automating Parallelism Discovery and Planning***
  Saturnino Garcia, Donghwan Jeon, Chris Louie, Sravanthi Kota Venkata, and Michael Bedford Taylor, University of California, San Diego

This work introduces pyrprof, which is a profiler for parallelism. It relies on the idea that potential parallelism is the ratio of work and the length of the critical path. Pyrprof ranks regions of code based on their parallelism potential and reports this information back to the user. The programmer can provide feedback to improve the accuracy of the system. Pyrprof will soon be publicly available. Contact: http://parallel.ucsd.edu/pyrprof.

- ***Checking Non-Interference in SPMD Programs***
  Stavros Tripakis and Christos Stergiou, University of California, Berkeley; Roberto Lublinerman, Pennsylvania State University

This work is like Lint for CUDA. It uses an SMT solver to determine if there are interferences in blocks separated by

__syncthreads. Contact: chster,stavros@eecs.berkeley.edu, rluble@psu.edu.

- ***Molatomium: Parallel Programming Model in Practice***
  Motohiro Takayama, Ryuji Sakai, Nobuhiro Kato, and Tomofumi Shimada, Toshiba Corporation

This framework allows easy parallel programming of platforms such as TVs. Mol is a C-like language that borrows characteristics from Haskell (functional and lazy evaluation). It describes the parallelism present. It is compiled to a bytecode. Atom describes the platform code (the target is mostly Cell). Contact: motohiro.takayama@toshiba.co.jp.

Posters below summarized by Rik Farrow (rik@usenix.org)

- ***DeNovo: Rethinking Hardware for Disciplined Parallelism***
  Byn Choi, Rakesh Komuravelli, Hyojin Sung, Robert Bocchino, Sarita Adve, and Vikram Adve, University of Illinois at Urbana-Champaign

The key concept here is that by creating disciplined software, problems in designing hardware will become simpler. They have written Deterministic Parallel Java as an exemplar. DPJ allows partitioning the heap into named regions, and language constructs define data dependencies between regions. Cache coherency becomes easier as the relationships between cache lines are spelled out in software, and message passing can be used for updating invalidated cache lines. Contact: denovo@cs.illinois.edu.

- ***Lock Prediction***
  Brandon Lucia, Joseph Devietti, Tom Bergan, Luis Ceze, and Dan Grossman, University of Washington

The authors wrote a trace generator that wrapped around the pthreads library to collect calls of lock acquisition functions. They investigated the PARSEC benchmark suite of multithreaded programs and performed offline analyses of the traces to predict the next thread to acquire a given lock. Using a handful of models for lock transitions, they tested the accuracy of each model against the traces for different programs. Each program had different lock acquisition characteristics, but past work has shown that accurate lock acquisition prediction does improve code performance. Their *most frequent transition* predictor model worked the best in general. Contact: http://sampa.cs.washington.edu.

- ***Resource Management in the Tessellation Manycore OS***
  Juan A. Colmenares, Sarah Bird, Henry Cook, Paul Pearce, and David Zhu, University of California, Berkeley; John Shalf and Steven Hofmeyr, Lawrence Berkeley National Laboratory; Krste Asanović and John Kubiatowicz, University of California, Berkeley

In Tessellation, applications and OS services are assigned to *Cells*, an abstraction that contains parallel software components and supplies resource guarantees. A two-level scheduler separates global resource allocations from local scheduling and resource usage. A policy service determines how resources are allocated to each Cell, and application-specific schedulers, such as Lithe, are responsible for scheduling threads within each Cell. At the global level, gang-level scheduling ensures that components within a Cell are available during scheduled runtime. Contact: yuzhu@eece.berkeley.edu.

- ***Processes and Resource Management in a Scalable Many-core OS***
  Kevin Klues, Barret Rhoden, Andrew Waterman, David Zhu, and Eric Brewer, University of California, Berkeley

ROS provides a new process abstraction, the manycore process (MCP). With MCP, there is only one kernel thread per process, rather than per thread, and cores provisioned to an MCP are gang-scheduled. Traditional system calls are asynchronous and non-blocking, and processes are notified before a core or other resource is revoked. Resources include anything that can be shared in a system: cores, RAM, cache, on-and off-chip memory bandwidth, access to I/O devices, etc. Contact: brho@eecs.berkeley.edu and yuzhu@eece.berkeley.edu.

JUNE 15, 8:30 A.M.–10:00 A.M.

Summarized by Chris Gregg (chg5w@virginia.edu)

- ***Dynamic Processors Demand Dynamic Operating Systems***
  Sankaralingam Panneerselvam and Michael M. Swift, University of Wisconsin—Madison

Sankaralingam Panneerselvam started by discussing the symmetric chip multiprocessor and why it does not support sequential workloads well. He then went on to show that the asymmetric chip multiprocessor satisfies diverse workloads well, but not as well as we would like. A dynamic multiprocessor, however, is flexible enough to adapt to the right configuration based on need. Dynamically variable processors lead to better performance with merging resources and shifting power and also lead to better reliability, because of the ability to have redundant execution.

Geoff Lowney asked why we need to reconfigure the OS. Panneerselvam said that an unexpected processor shutdown can lead to thread execution stopping (in the case of a lock, for instance) or other stalls. He then described Linux HotPlug, which allows dynamic addition or removal of a processor. This allows for partitioning and virtualization, and for physical repair of the processor. It can be used for long-term reconfigurations, which assumes that the processor will never come back online, and all relevant systems are notified.

Dan Grossman asked, "When you say 'short-term reconfiguration,' what time frame are we talking about?" Panneerselvam answered, "Milliseconds." Performance is good for virtualization, but too slow for rapid reconfiguration. Next, Panneerselvam described the "processor proxy," which is a fill-in for an offline processor. The proxy does not actually execute threads, but ensures that everything else continues. Proxies are not a long-term solution, but if the reconfiguration is long-term, it is better to move to a stable state. To do

this, a "deferred hotplug" happens, which means that a CPU that is currently proxied is removed. A "parallel hotplug" can also happen, which is the reconfiguration of multiple CPUs. These methods provide greatly improved performance.

Timothy Roscoe asked, "Why is Linux the right OS to try this in? How much of this is about monolithic kernels?" and there was a long discussion about the role of the monolithic kernel. Panneer Selvam said that if we assume the virtual case, or a hypervisor, we want a number of things added to the OS in order to handle it. The OS wants to know about the changes, and we can implement those changes in Linux, in the monolithic kernel. Monolithic kernels aren't going away soon.

- ***Design Principles for End-to-End Multicore Schedulers***
  Simon Peter and Adrian Schüpbach, ETH Zurich; Paul Barham, Microsoft Research, Cambridge; Andrew Baumann, ETH Zurich; Rebecca Isaacs and Tim Harris, Microsoft Research, Cambridge; Timothy Roscoe, ETH Zurich

Simon Peter described the scheduler in Barrelfish, an experimental operating system. He started by asking why having two applications, one CPU-bound and one a barrier application, are a problem for OpenMP, in particular on a 16-core system. The barrier application shows decreased performance as the number of barrier threads increases. This is because of the increasing cost to execute the barriers. This situation works fine for a small number of threads, but eventually the performance drops significantly. Their approach mitigates this with gang scheduling and smart core allocation. Peter proposed an end-to-end approach, involving all components that can cut through classical OS abstractions, focusing on OS and runtime integration.

Peter then described the five design principles Barrelfish implements. First, he discussed time-multiplexing cores that offer real-time quality of service for interactive applications. David Patterson said, "There is a possibility in the future that we cannot turn on and off cores at will, because of power issues. Is it still worthwhile to use time-multiplexing instead of space-multiplexing?" Peter answered that actually, this is a perfect case for time-multiplexing, because you might have to time-multiplex the cores that you do have access to.

Peter then discussed scheduling at multiple timescales. He described the need for a small overhead when scheduling, because synchronized scheduling on each time-slice won't scale. This is implemented in Barrelfish with a combination of techniques, including long-term placement of applications on cores, medium-term resource allocation, and short-term per-core dispatch. David Patterson then asked, "What is the problem this is trying to solve?" Peter replied that they are trying to decouple things so we don't have to reschedule all the time. Barrelfish has phase-locked gang scheduling, which decouples schedule synchronization from dispatch. There may be a future re-sync necessary, but this happens at coarse-grained time scales.

Peter then outlined the "system knowledge base" in Barrelfish, which contains a rich representation of the hardware in the system. The OS and applications use this database. David Patterson asked, "Why did you go with a system knowledge base, which seems like a central bottleneck? Why didn't you make it a runtime database?" Peter answered that the hardware discovery information, boot-time micro-benchmarks, etc., go into the database. The data can be comprehensively queried, and the applications can use the database effectively. The centralized database was their first attempt, and it will be improved in the future.

JUNE 15, 10:30 A.M.–12:30 P.M.

Summarized by Rik Farrow (rik@usenix.org)

- ***OoOJava: An Out-of-Order Approach to Parallel Programming***
  James C. Jenista, Yong hun Eom, and Brian Demsky, University of California, Irvine

Jim Jenista described how they had created a version of Java that can add parallelism to serial programs. In this work, they added a single language construct, the reorderable block, or *rblock*, that designates portions of code that can be executed out of order. Rblocks can be executed as soon as all dependencies are satisfied.

The OoOJava compiler builds graphs between parent and child blocks and safely determines all data dependencies automatically. Jenista admitted that their implementation has several limitations, including a single exit point from each rblock. Dan Grossman immediately asked about exceptions, and Jenista answered that they use a subset of Java with no exceptions. He went on to describe a simple code example with two rblocks and explained the tree of dependencies that would be created, then walked, during execution. This graph shows that heap dependencies are properly handled, that all writes to a memory location occur in the same order.

Someone asked about virtual functions, and Jenista replied that they make a summary of all possible methods and combine them. Another person wondered if they had threads. Jenista answered that their subset has no threads, exceptions, global variables, or reflections. But, given a serial program, OoOJava creates a parallel program out of it.

David McCool asked how many lines of code this required, and Jenista said several thousand. They convert Java into C code that is compiled, resulting in a decent speedup. The code is available at http://demsky.eecs.uci.edu/compiler.php and includes other research features as well. David Padua wondered what happens if the compiler fails, and Jenista answered that the compiler reports that to you and suggests changes.

- ***User-Defined Distributions and Layouts in Chapel: Philosophy and Framework***
  Bradford L. Chamberlain, Steven J. Deitz, David Iten, and Sung-Eun Choi, Cray Inc.

Brad Chamberlain described Chapel, a new language that supports parallelism. Chapel is part of the DARPA-led High Productivity Computing Systems program. The language is designed to improve the programmability, robustness, and performance of parallel programs and targets both multicore and commodity cluster systems. You can download the source code from http://sourceforge.net/projects/chapel/.

Parallelism and data locality are driving concerns in Chapel. Chapel includes notation for arranging data in arrays and how data parallel operators should be implemented. McCool asked about their strategy for vector instructions, and Chamberlain responded that they haven't created vector compilers, but generate C code.

Chamberlain then explained domains and domain mappings. Domains takes lists of indices, and domain maps specify how the data will be accessed—for example, with a blocking factor or by tiling. An example mentioned a *zippered* domain map, and someone asked what "zippered" meant. Chamberlain explained that you would use zippering to suggest to the compiler which iterator to use when you have two domains with different layouts.

Chamberlain said that Chapel includes a user-defined domain-map framework and that at Cray they use this framework themselves. They don't want to have an unfair advantage using a tool that is publicly funded. McCool asked if the compiler can convert nested multiple arrays, and Chamberlain answered, not currently, but there are default domain maps you can use to support this explicitly. Dan Grossman asked if using Chapel avoids static analysis, and Chamberlain said that you still have to do this yourself. Grossman said that you expose this, but Chapel does not understand it, and Chamberlain agreed. He said that you want to implement the right domain maps whenever possible.

One goal of Chapel is not to impose arbitrary limitations. They do want to support separation of roles, with parallel experts writing domain maps and others using them. Chapel does support both CPUs and GPUs. They have compared Chapel to CUDA and gotten the same performance using a smaller code base. Sarita Adve asked about loads and stores, and Chamberlain responded that they support normal C indexing and that memory consistency is incredibly relaxed. The programmer is responsible for arranging copying data between main memory and the GPU.

Grossman asked if the goal of Chapel is to become popular or develop new language structures, and Chamberlain answered that either would be satisfactory. The main goal is to make users more productive. Grossman asked about status. Chamberlain said that performance is not good enough yet, but please try Chapel and provide feedback. You can find the slides for this presentation at http://www.usenix.org/events/hotpar10/tech/slides/chamberlain.pdf.

- ***On the Limits of GPU Acceleration***
  Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Efe Guney, and Aashay Shringarpure, Georgia Institute of Technology

Richard Vuduc started his talk with a quote: always compare your results with scalar, unoptimized Cray code, as this will make your code look good. He then said that his paper was more of a survey, perhaps a story. The story begins with Scott Klasky posing a question: should I port my application to GPUs? A quick literature search turns up amazing speedups, 30–100 times faster than running on a modern CPU. What Vuduc and his fellow researchers found was something very different.

Vuduc pointed out that current GPUs are bandwidth-bound, as they sit on the PCIe bus. A related issue has to do with memory access patterns. McCool asked if working-set size matters, and Vuduc said that has some influence. Even with the GPU on the same die as the CPU, there could still be bandwidth issues. Patterson asked if he was suggesting a second memory, and Vuduc pointed out that you might need to keep GPU memory even in the on-die version.

The bottom line is that with code properly tuned to run on a multicore system, like a Nehalem, the big exciting differences fade away. The authors tried three different scientific computations: (1) iterative sparse linear solvers, (2) sparse Cholesky factorization, and (3) the fast multipole method. Geoff Lowney asked how much work was involved in tuning, and Vuduc said that someone spent perhaps one month of work, producing roughly twice the number of lines of code, in tuning one application. Lowney then wondered if the NVidia GPU code also represented tuned code and Vuduc said they were well tuned, with NVidia's cooperation.

In the sparse matrix and fast multipole methods, the issue is clearly bandwidth related. Andrew Bauman asked if pipelining would help, and Vuduc said that a student is working on that. By tuning code, they found that a multiple core version on Nehalem was only 10% slower than a dual GPU version. In summary, one GPU is roughly equal to one CPU. If you look at power, CPUs are better. Someone asked why it was easier to gain so much speedup on GPUs? Vuduc answered that it isn't really, that it took an equal amount of effort to tune and prepare code for either GPU or CPU.

JUNE 15, 2:00 P.M.–4:00 P.M.

Summarized by Romain Cledat (romain@gatech.edu)

- ***Gossamer: A Lightweight Programming Framework for Multicore Machines***
  Joseph A. Roback and Gregory R. Andrews, The University of Arizona, Tucson

Gossamer is a framework for annotating existing applications to make them parallel. Roback first presented the 15 annotations that compose Gossamer. The annotations are meant to encompass as large a domain as possible and support task spawning through constructs such as fork,

parallel, divide and replicate, memory synchronization with join, barrier, atomic, buffered, copy, ordered, shared, and the map-reduce paradigm. Roback then illustrated the annotations with a variety of well-known examples. In the n-queens problem, he showed how fork and join could be used. In bzip2, the presenter also demonstrated the "ordered" keyword, which allows a serialized join in the order of spawning.

Roback briefly described the source-to-source translator that is used to compile down the annotations and generate bookkeeping artifacts. For example, the translator tries to limit the number of locks required to enforce "atomic" sections by finding the best middle ground between one global lock and one lock per variable.

Roback then described the runtime involved. The application-level threads are referred to as "filaments" and are stackless and stateless, making them extremely lightweight. The filaments share the stack of the thread they are running on. A member of the audience asked if, once placed on a thread, a filament had to run till completion. Roback said yes, at this time, as the threads are stackless, but the authors are exploring medium-weight threads that could be interrupted. David Padua asked about the producer-consumer paradigm and Roback said this was also future work. Recursive and task filaments are scheduled in a round-robin fashion: iterative filaments are scheduled in groups to maximize cache locality, and domain decomposition filaments are scheduled statically with one filament per processor.

Results were presented that demonstrated the very low overhead of the system. The super-linear speedup in the matrix-multiplication benchmark is due to the fact that when the benchmark runs on two sockets, it has a larger L2 cache. Results also showed Gossamer comparing positively to Cilk and OpenMP in most situations.

In conclusion, Gossamer is a simple portable framework and the translator is available as a stage in the compilation process and can therefore be simply plugged into GCC or ICC.

David Patterson asked if they were thinking about trying out larger applications (like the Dwarfs and the implementation presented at this year's HotPar). Roback answered that the goal was to try to fit as many applications as possible. David Padua asked why OpenMP was so much slower at times than Gossamer, since the approach seemed similar. It's because the task implementation in OpenMP is currently not very good.

- ***Reflective Parallel Programming: Extensible and High-Level Control of Runtime, Compiler, and Application Interaction***
  Nicholas D. Matsakis and Thomas R. Gross, ETH Zurich

Matsakis presented the concept of "reflective parallelism," which he describes as a program's ability to reason about its own schedule at runtime. Consider, for example, two tasks "A" and "B." Questions that could be answered with reflective parallelism are: "Do A and B always run in parallel?" and "Must A finish before B starts?" The results from queries should return results that hold true for all executions and the program should be able to dynamically modify the schedule by adding scheduling constraints. Matsakis stated that reflective parallelism could be used for many things, from schedule visualization to testing frameworks to data-race detection. In this paper he focused on data-race detection.

Matsakis then exposed the big problem with current threads: they construct their schedule through primitives such as "start," "join," and "wait," but the schedule is therefore never explicit until after the whole program has executed. Even after the program has run, it is nearly impossible to analyze the schedule and come up with assertions that are always true. Reverse-engineering the program to build the schedule is also risky.

Matsakis then introduced his answer to these problems: make the schedule a first-class entity in the program with the use of intervals where their use can express the schedule through declarative methods. The three concepts captured by the model are: (1) intervals that represent an asynchronous task or group of tasks; (2) points that represent the start and end of intervals (the point right before an interval and right after) on which "HappensBefore" relationships can be specified; and (3) locks that can be held by intervals to specify a constraint without imposing an order. Alexandra Federova asked how this was different from TBB, and Matsakis responded that although a task-graph existed in TBB, it was more low-level with reference counts and was thus not a first-class entity.

Matsakis then briefly described the scheduling model where a "ready()" method expresses to the runtime that an interval is ready to run. "HappensBefore" relationships can be added dynamically, but they cannot be removed, which guarantees monotonicity and makes scheduling easier.

Finally, Matsakis defined how reflection can be used to specify "guards" on data objects. Guards can evaluate a condition based on information gleaned through reflection to determine whether the object they are guarding can be accessed. Many of these conditions can be known at compile time, but even if they cannot, they can be quickly evaluated at runtime and warn the user of any data-race.

In summary, the intervals framework, available at http://intervals.inf.ethz.ch, allows users to specify access conditions using information reflected back about the schedule.

Geoff Lowney asked how the system handles the case where there is no guard on an object. Matsakis responded that the framework mandates a guard for all fields. Another person asked how to know if this is the right way to proceed. Matsakis said it was a tough question to answer but that they had tried this model in undergrad classes with success. Finally, a member of the audience asked how many of the checks were dynamic. Matsakis answered that many

checks could be done statically and, as is the case for most type systems, some small restructuring of the program can expose even more static checks.

- ***Task Superscalar: Using Processors as Functional Units***
  Yoav Etsion, Barcelona Supercomputing Center; Alex Ramirez, Barcelona Supercomputing Center and Universitat Politècnica de Catalunya; Rosa M. Badia, Barcelona Supercomputing Center; Eduard Ayguade, Jesus Labarta, and Mateo Valero, Barcelona Supercomputing Center and Universitat Politècnica de Catalunya

In this talk Etsion presented the idea of extending out-of-order instruction pipelines to tasks to aid in exposing the operations that can execute in parallel and manage data synchronization. Indeed, for many years out-of-order pipelines have been managing parallelism in a sequential stream of instructions. Although ILP does not scale well, due to the problems of building a large instruction window (difficulty with building a global clock, as well as the limited scalability of dependency broadcasts) and the unpredictability of control-paths, Etsion believes that out-of-order task parallelism may work better.

The presenter then moved on to explain the StarSS programming model, where tasks are modeled as abstract instructions. A master thread spawns the various tasks encountered, which are dispatched to the worker processors. A runtime dynamically resolves dependencies and constructs a task-graph. It is important to note that the task-graph can get very complicated very quickly but that StarSS can build it and exploit it efficiently.

The need to do the task-decoding and scheduling in hardware is due to the high latency of software (between 700ns and 2.5us).

The model of execution is very similar to that of out-of-order instruction execution: tasks are decoded and pushed to reservation stations. Data dependencies are taken care of in the same way as for instructions. Etsion showed results that demonstrated that parallelism could be uncovered in many scientific applications.

Etsion explained that task parallelism will scale more than ILP, for a variety of reasons. Firstly, broadcasts do not have to be used, since the latencies involved are much higher. Dependencies can therefore be dealt with using point-to-point communication, which is much more scalable. Secondly, there is no need for a global clock. Thirdly, the multiplex reservation stations allow multiple tasks in the same data structure, making the representation much more compact. Tasks also are not speculative, although the authors are looking at task predication.

As future work, the authors wish to exploit locality-based scheduling and also to gather tasks using a similar kernel and package them off to a GPU. They also wish to explore which instruction-level optimizations can be applied.

David Padua asked if tasks can interrupt each other. At this point they cannot, but nothing verifies that this is the case. Another audience member asked how energy-efficient the model is. It is difficult to predict, although it seems to be more efficient than having a dedicated big core decode and schedule the tasks.