HPPC 2009 Panel: Are many-core computer vendors on track?

Martti Forsell[1], Peter Hofstee[2], Ahmed Jerraya[3], Chris Jesshope[4], Uzi Vishkin[5], and Jesper Larsson Träff[6]

[1] VTT – Technical Research Centre of Finland
Oulu, Finland
[2] IBM Systems and Technology Group
USA
[3] CEA - LETI MINATEC
Grenoble, France
[4] University of Amsterdam
Amsterdam, The Netherlands
[5] University of Maryland Institute of Advanced Computer Studies (UMIACS)
Maryland, USA
[6] NEC Laboratories Europe, NEC Europe Ltd.
St. Augustin, Germany

1 Introduction

The last session of the HPPC 2009 workshop was dedicated to a panel discussion between the invited speakers and three additional, selected panelists. The theme of the panel was originally suggested by Uzi Vishkin, and developed with the moderator. A preamble was given in advance to the five panelists, and provoked an intensive and determined discussion. The panelists were given the chance to briefly summarize their view- and standpoints after the panel.

Panelists: Martti Forsell, Peter Hofstee, Ahmed Jerraya, Chris Jesshope, Uzi Vishkin.

Moderator: Jesper Larsson Träff.

2 Preamble: Background and issues

The current proliferation of (highly) parallel many-core architectures (homo- and heterogeneous CMP's, GPU's, accelerators) puts an extreme burden on the programmer seeking (or forced) to effectively, efficiently, and with reasonable portability guarantees utilize such processors. The panel will consider whether what many-core vendors are doing now will get us to scalable machines that can be effectively programmed for parallelism by a broad group of users.

Issues that may be addressed by the panelists include (but not exclusively): Will a typical computer science graduate be able to program mainstream, projected many-core architectures? Is there a road to portability between different types of many-core architectures? If not, should the major vendors look for

other, perhaps more innovative, approaches to (highly) parallel many-core architectures? What characteristics should such many-core architectures have? Can programming models, parallel languages, libraries, and other software help? Is parallel processing research on track? What will the typical computer science student need in the coming years?

3 Martti Forsell

The sequential computing paradigm formulated in the 50's has been tremendously successful in the history of computing. The main reason for this is the right type of abstraction capturing the essential properties of the underlying machine and providing good portability between machines with substantially different properties. The idealized properties of the computational model – single cycle access time and sequential operation – can be emulated well enough even with speculative superscalar architectures and considerably deep memory hierarchies applying paging virtual memory. At the same time, sequential computing is easy to learn and use, there is a thorough theory of sequential algorithms, and efficient teaching in universities. Synchronization of subtasks of originally parallel computational problems is trivial due to deterministic sequential execution. Performance enhancement techniques, including exploitation of low-level parallelism, speculations, and locality exploitation, are well-known and linked to the paradigm.

Parallel computers introduced in the 60's and the related parallel computing paradigms have had a totally different reception than sequential ones, having doomed them to marginal uses except in high-performance computing. During this decade the situation has, however, changed totally with the arrival of multicore processors. Parallel computing is here to stay with no way back to sequential machines any more. This is because of power density problems preventing exponential growth of clock frequencies for microprocessors. Unfortunately current approaches to parallel computing (e.g. SMP, NUMA, CC-NUMA, vector computing, and message passing) are weak making the whole paradigm poor. Namely, the abstraction of the underlying parallel machinery is too low and inappropriate: A programmer is forced to take care of mapping of functionality, partitioning of data, and synchronization. In the message passing model, one even needs to take care of low-level sending and receiving messages between processes. As a result, programming is difficult and error-prone. The portability between machines with different properties is often limited and easily leads to a need to rewrite the entire software for the new machine. The generality of the theory of parallel algorithms is severely limited by architecture dependency of current solutions, and teaching is virtually nonexistent at elementary level even in universities. Execution of subtasks is asynchronous and the cost of doing an explicit barrier synchronization is easily hundreds or thousands of clock cycles. This severely limits the applicability of current approaches and effectively rules out fine-grained parallel algorithms. Performance enhancement techniques are not particularly innovative nor well-linked to thread-level paral-

lel execution since they are mostly copied from sequential computing, whereas efficient techniques for parallel computing, including co-exploitation of ILP and TLP, concurrent memory access, and multioperations, are missing.

Historically speaking, the trends of architectural approaches towards increasingly parallel and complex machines seem to point towards more difficult programmability. There exists, however, approaches to parallel computing, e.g. vector computing and PRAM, that are easy to program and therefore would solve most of the problems listed above. While the somewhat successful vector computing approach is limited to vectorizable algorithms due to an inability to exploit control parallelism, the more flexible and theoretically beautiful PRAM has been widely considered impossible to implement. According to our implementation estimation studies on advanced parallel architectures this conception appears to be wrong. Therefore, to address programmability and applicability issues, we are developing CMP architectures realizing the PRAM model and related application development methodology. We have just started a project to build our first FPGA prototype. Interestingly we are not alone, two out of five panel participants are doing research in this direction.

4 Peter Hofstee

Driven by the need to deliver continuous performance improvements without disturbing the existing code base, all major high-performance CPU vendors have opted for shared-memory multi-core/multi-thread architectures. With this approach, existing applications with a modest amount of concurrency still benefit from the larger caches, increased memory capacity and bandwidths, and increased I/O capabilities that a next-generation processor provides. The need to provide incremental performance improvements on all applications also is forcing vendors to continue to make modest improvements to per-thread performance, and this limits their ability to achieve the best possible power efficiencies for concurrent applications. All major vendors now integrate the memory controllers. Integration of I/O and graphics is likely to be next leading to more heterogeneous multi core processors. Large machines can be expected to be built as clusters of these SMP nodes, though it is likely that even across these clusters address spaces will be increasingly unified and shared.

Given this type of hardware, the SMP node memory looks more or less "flat", i.e. access latencies to memory are not significantly dependent on which core on the chip is accessing what memory attached to the node. Even if memory is shared across the cluster, latency and bandwidths are substantially different for memory attached to the local node and memory attached to remote nodes.

In order to prepare students for the future we need a fundamentally new approach to the way students are taught. The fundamentals that drive algorithmic efficiency on today's and future hardware are:

Classical notions of complexity – the total number of operations (memory accesses, algorithmic operations etc.) as taught today.

Concurrency – A more concurrent algorithm of the same overall complexity is more valuable.

Predictability – An algorithm in which data references and control flow are predictable is more valuable (data parallelism can be regarded as a form of data and control flow predictability).

Locality – An algorithm with better data and control flow locality is more valuable.

Each of these notions of complexity leads to fundamental transformations of the algorithms that are beyond a compiler's ability to perform automatically. Language designers should therefore build on these fundamental notions of algorithmic efficiency while preserving conciseness of expression and semantic clarity. Of course libraries can help those who program computers, not every driver has to be a mechanic, but we should teach computer science students the fundamentals, as we will need many skilled people to restructure our code base such that it can be efficiently targeted at today's and future highly parallel processors.

5 Ahmed Jerraya

The shift from the single processor to heterogeneous multiprocessor architectures poses many challenges for software designers, verification specialists and system integrators. The main design challenges for multi-core processors are: programming models that are required to map application software into effective implementations, the synchronization and control of multiple concurrent tasks on multiple processor cores, debugging across multiple models of computation of MPSoC and the interaction between the system, applications and software views, and the processor configuration and extension.

Programming an MPSoC means to generate software running on the MPSoC efficiently by using the available resources of the architecture for communication and synchronization. This concerns two aspects: software stack generation and validation for the MPSoC and communication mapping on the available hardware communication resources and validation for MPSoC. Efficient programming requires the use of the characteristics of the architecture. For instance, a data exchange between two tasks mapped on different processors may use different schemes through either the shared memory or the local memory of one of these processors. Additionally, different synchronization schemes (polling, interrupts) may be used to coordinate this exchange. Furthermore, the data transfer between the processors can be performed by a DMA engine, thus permitting the CPU to execute other computation, or by the CPU itself. Each of these communication schemes has advantages and disadvantages in terms of performance (latency, throughput), resource sharing (multitasking, parallel I/O) and communication overhead (memory size, execution time). The ideal scheme would be able to produce an efficient software code starting from high-level program using generic communication primitives.

For the design of classic computers, high-level parallel programming concepts (e.g. MPI) are used as an Application Programming Interface (API) to abstract

hardware/software interfaces during high level specification of software applications. The application software can be simulated using an execution platform of the API (e.g. MPICH) or executed on existing multiprocessor architectures that include a low level software layer to implement the programming model. In this case the overall performances obtained after hardware/software integration cannot be guaranteed and will depend on the match between the application and the platform. Unlike classic computers, the design of heterogeneous MPSoC requires a better matching between hardware and software in order to meet performances requirements. In this case, the hardware/software interfaces implementation is not standard; it needs to be customized for a specific application in order to get the required performances.

Therefore, for this kind of architectures, classic programming environments do not fit: (i) high level programming does not handle efficiently specific I/O and communication schemes, while (ii) low level programming explicitly managing specific I/O and communication is time consuming and error-prone activity. In practice, programming these heterogeneous architectures is done by developing separate low level codes for the different processors, with late global validation of the overall application with the hardware platform. The validation can be performed only when all the binary software is produced and can be executed on the hardware platform. Next generation programming environments need to combine the high level programming models with the low level details. The different types of processors execute different software stacks. Thus, an additional difficulty is to debug and validate the lower software layers required to fully map the high-level application code on the target heterogeneous architecture.

6 Chris Jesshope

Are manufacturers doing enough is perhaps the wrong question. We should be asking whether they did enough to manage the entirely predictable shift from sequential computing to parallel computing as a direct consequence of the power wall. And the answer is probably no; we are unprepared.

Users have come to expect universality; sequential code works on all architectures either using source-code or binary-code compatibility and this is what they now expect from multi-cores and concurrent heterogeneous systems. Concurrency however, introduces all sorts of difficult problems, including the mapping and scheduling of work, races, deadlocks and fairness issues, etc. Ideally applications engineers (programmers) should not be exposed to the latter.

A key issue therefore is whether we can separate algorithm-engineering issues from concurrency engineering ones. Algorithm engineering does not usually require concurrency, just a deterministic parallel implementation. A major problem is in dealing with synchronisation state, it complicates algorithm engineering unnecessarily and constrains the problem mapping. It is not strictly necessary and there are approaches, which aim to provide such a separation of concerns. These are emerging technologies however, and are academic rather than commercial.

A further issue is whether we can program in a manner which is independent of the scale of concurrency to which the the code will eventually be targeted, i.e. can we code once and run anywhere, in order to have code portability across different classes of target architecture. Again there seems to have been little work moving us in this direction. It requires abstract concurrent programming models that allow the capture of maximal concurrency and, ideally, also capture locality. A typical approach is to take code and to parallelise it to given target with a given granularity of parallelism. However, when you change the parameters or the target it needs to be rewritten. The alternative is to program at the finest grain possible and then sequentialise the code automatically when a target is chosen. In this way the same code can be efficiently executed on any target and the procedure of sequencing parallelism is a rather trivial one compared to parallelisation. Again work is being carried out in this area but is also academic.

Models that are maximally concurrent but abstract (e.g. SVP) and coordination languages that allow this separation of concerns (e.g. S-Net) have been developed in the EU AETHER project, which has taken a 10-year-out view on programming highly concurrent and heterogeneous systems (see: `http://www.aether-ist.org/`).

7 Uzi Vishkin

Hardware vendors have been forced into replacing the serial paradigm that has worked for them well for decades by parallel computing based on many-core architectures. To date, no commercial easy-to-program general-purpose many-core machine for single-task completion-time has been available. In fact, several decades of parallel architectures have been able to produce only rather limited success stories. A 2003 NSF Blue Ribbon committee effectively declared their programming a "disaster area" by noting that to many programmers it is "as intimidating and time consuming as programming in assembly language". Hardware vendors need to reproduce the serial success for many-cores. Adopting, without significant modification, the same parallel architectures would instead drag mainstream computing into the same disaster area.

Customers buying a computer interact with its software, but their link to the hardware is indirect, by nature. However, the cyclic process of hardware improvements leading to software improvements, which lead back to hardware improvements and so on, known as the software spiral, facilitated for many years a direct link between customers and hardware. Hardware designers could directly serve their customers by helping to run serial code faster. Alas, the software spiral is now broken. Consequently, getting application software developers (ASDs) to switch to the emerging generation of many-core systems has become much more critical to serving these customers. However, the incentive to develop software for the new machines has decreased considerably. Code development and maintenance is much more expensive, as initial development time is higher and code is more error prone. Not only that the investment is higher, the returns on it are much riskier: even if machines continue to support the current develop-

ment platform, some hard-to-predict future upgrades may offer new options for optimization of performance, allowing competitors to develop better software, at a lesser cost, by just adopting a wait-and-see approach. Thus, computer designers need to understand the legitimate concerns of software developers and do what they can to "woo" them.

The explicit multi-threading (XMT) approach `www.umiacs.umd.edu/users/vishkin/XMT/` could affect the above discussion in two ways. First, it affirms concerns that hardware improvements that may significantly reduce investment in code development by just waiting till they are installed are indeed possible. The second way is that incorporation of the hardware upgrades that XMT suggests, could make it possible to support the broad family of PRAM algorithms, greatly alleviating current concerns about ease-of-programming. Moreover, every person majoring in CS should be able to program the commodity many-core system of the future. Teachability of XMT programming has been demonstrated at various levels from rising 6th graders to graduate students, and students in a freshman class were able to program 3 parallel sorting algorithms.