Challenges of parallel processor design

M. Forsell and V. Leppänen and M. Penttonen

May 12, 2009 21:48

Abstract

While processor speeds have grown, also the gap between the speed of processing and the speed of accessing data has grown. Therefore the speed of the processor cannot be used. As a reaction, processor industry has started to build more processor cores on a chip, but there is no easy way to utilize multiple processors. In this work we study alternative multicore processor designs that could efficiently run parallel programs.

1 What Moore said?

Very soon after the success story of microelectronics had started, G. Moore published a forecast [13] that was later to be called the "Moore's law". By development from year 1959, when a circuit consisted of one electronic component, to year 1965, when 50 components could be packed on a circuit, he bravely forecast that in 1975 one could perhaps pack economically as much as 65000 component. In other words, in 16 years the packing density would grow 2^{16}-fold. In 2007 the packing density was almost 5 billions, or about 2^{32}-fold. Hence, in 48 years the packing density did not grow 2^{48} fold but still the "law" can be stated in a milder form: "packing density doubles every 18 months".

In recent years the "Moore's law" got more popular formulations like "the power of pc's doubles every 18 months" or alike. Similar "laws" have been presented for the growth of the bandwidth of data communication. Can we trust on such "laws"? Even if we can always hope for the revolutionary inventions by scientists, such as quantum computation, we must be aware of the physical constraints of our world. An electronic component cannot become smaller than an atom (or an elementary particle). To transport information from one place to another needs some time. At very high packing density and high clock rate, heat production becomes a problem. Electrical wires on circuits cannot be radically thinner than what we have got now, or quantum effects start to appear.

The current packing density already has lead the processor industry to a problematic situation: How to get optimal computational power from the chip? How to get data at right time at right place so that the computation is not delayed by latencies? Overheads of the memories and the time of moving data over physical distances imply latencies that are about a hundred times more than the time required by the instruction itself.

This has lead to complicated caching, which should be called art rather than science. As computation depends on data, also compilation of programs has become art, when one tries to guess the branch, how the computation continues, in order to start fetching data as early as possible. It has been possible to build more and more components on a chip, but it is difficult to speed up a single computation. The computer industry has kind of "raised up hands" by starting to build multiple processor "cores" on one chip without clearly seeing, how to use them.

In principle multicore chips solve the "von Neumann bottleneck" problem, as all data need not be processed at the same processor core. The big new problem is that can we utilize multiple cores. If there are two or four cores, we can give detailed instructions, what to do in each core. But if the number of cores grows, can the programmer use them optimally? Anyway, programming is difficult enough without new factors to optimize. If the multiple cores are not used in a systematic and efficient way, one can easily end up with a program that is slower than the sequential unicore program.

2 Theoretical basis of general purpose parallel computing

The advantage of high processing speed is lost, if the data to be processed is not present. All data cannot always be present and due to distance and hardware overheads, fetching data takes enormous time in comparison with processing speed. Without losing the generality of the computation, we cannot assume that at successive steps of computation only "local" data is used. In the worst case, each sequential step of the computation could need data from the "remotest" part of the memory. One can perhaps predict, what data the next few instructions need and prefetch them to a cache, but the bigger the latency gap grows, the harder is the prediction. This is a hard fact: it is useless to just speedup a processor, if we cannot guarantee that it can do something useful.

Parallel processing offers a solution. If there are many independent threads of computation available, instead of waiting for data the processor can move (even literally) to process another thread. While other threads are processed, the waited data hopefully comes, and the processing of the waiting thread can be continued.

However, some questions rise:

1. Can we find enough independent parallel threads in the computation so that latency time can be meaningfully used?

2. Can the data communication machinery of the computer serve all parallel threads?

3. Can the swapping between threads be implemented so that efficiency is not lost?

4. Can all this be done so that programming does not become more difficult than before?

The first question is algorithmic and the second one concerns the hardware architecture. The last two questions are not theoretically as fundamental, but the idea of parallel threads lives or dies depending on how successful we are at these questions. In short, the condition of a successful model of computation is that algorithm design should be possible at a high level of abstraction, and still algorithms can be automatically compiled to programs that run efficiently on hardware.

The theory of parallel algorithms, in particular the PRAM (Parallel Random Access Machines), see [9] answers positively to the first question. A lot of parallelism can be found in computational tasks. It also answers quite positively to the last question. Even programming for the PRAM model is different from programming for the sequential RAM model, it is not more difficult as soon as we can get rid of the fixed idea of sequential programming.

Questions 2 and 3 are more difficult to answer. Based on the analysis [2] of the situation in nineties, the PRAM model was deemed to be unrealistic. PRAM assumes that an unbounded number of processors can run synchronously from instruction to instruction, even if some instructions require data from an arbitrary memory location, and many processors may be writing to the same memory location. Parallel processors cannot get rid of the latencies of the physical world, but as it was hinted above, parallelism offers a way to use the waiting time meaningfully. The idea of *parallel slackness* was proposed by Valiant [14]. If the parallel algorithm uses sp virtual processors, where s is the slackness factor and p is the number of physical processors, each virtual processor can use only fraction $1/s$ of the physical processor time, i.e. the computation of the virtual processor proceeds every s'th step of time. If s is not smaller than the latency l, the computation of the virtual processor proceeds at full speed with respect to the amount of physical processors. In other words, the slackness factor can be seen to decrease the clockrate of a virtual processor executing a thread by the factor s compared to the clockrate of the physical processor (and thus making the processor-memory speed gap to vanish).

In principle, the slackness solves the latency problem, but it implies a demanding data communication problem. All the time all p processors may want to read data anywhere in the computer, and each read instruction lives for the slackness time s in the routing machinery. Hence, there may be ps data packets in the internal network of the computer. As processors have nonzero volume, the distances grow at least with the qubic root of the number of processors, i.e. $s \geq l \in \sqrt[3]{p}$. In practice, the topology and many other properties of the network determine, how much slackness is needed, but the bandwidth requirement ps is unavoidable. In 2-dimensional sparse torus [12], for example, p processors are connected by a $p \times p$ mesh (or torus), where the processors are on the diagonal. In this structure the diameter of the network is p, the latency is $2p$ and latency $s = 2p$ can be used, because the bandwidth of the network is $2p^2$.

3 Parallel processor designs

In order to keep the programmer's model of computation as simple as possible, we want to see the parallel computer as in Figure 1

A vector compaction program (eliminating zero elements) for this machine would look like

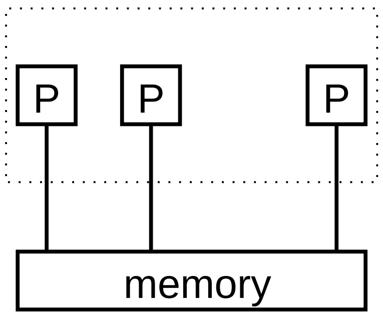

Figure 1: Parallel Random Access Machine

```
proc compact(A)
for i=0..n-1 pardo
  if A[i]=0 then C[i]=0 else C[i]=1
E=prefix-sum(C)
for i=0..n-1 pardo
  if A[i]<>0 then B[E[i]]=A[i]
return B
```

In this program, function `prefix-sum(C)` computes vector E where E[i]=C[0]+C[1]+ . . . +C[i]. If we assume that `prefix-sum` can be computed in time $O(1)$, also `compact` can be computed in time $O(1)$.

In Figure 1, there is a dotted box around the processors, hinting that those processors should be built on a chip. We are now taking a closer look, how could such a multiprocessor chip be built.

3.1 Paraleap - PRAM on Chip

Vishkin's team [15] has built a processor, whose simplified schematic is shown in Figure 3.1. Master Thread Control Unit (MTCU) has a central role in scheduling the threads. Whenever there are independent parallel threads waiting for being processed, they ask their turn from MTCU. Whenever a thread is completed at a processor P, it is reported to MTCS. In the current version of the Paraleap, the number of processors, or more exactly Thread Control Units (TCU), is 64. However, TCU's are not fully indendent but clusters of 16 TCU's share some functional units.

Prefix Sum Unit (PSU) is another important component in Paraleap. It can be used in programs for such purposes as in the vector compaction program. However, the most important usage is the scheduling of threads to TCU's. When TCU's run and complete threads in parallel, PSU calculates, which is the next thread to be started.

Processors are connected to the shared memory by the internal network, which is a mesh of trees in Paraleap. There are 8 caches between the network and the main memory. Processors share a set of Registers (Reg).

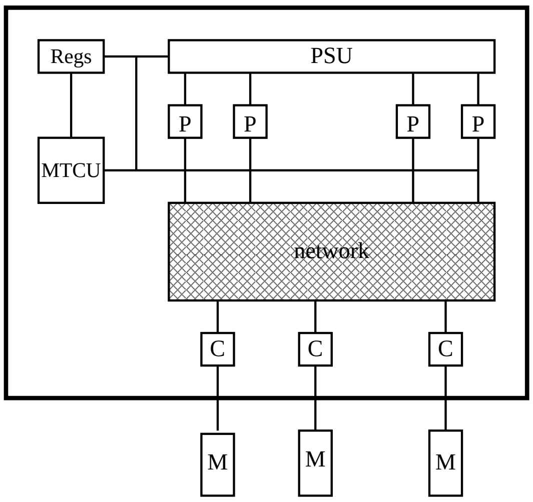

Figure 2: Structure of the Paraleap processor

Paraleap has been implemented on Field Programmable Gate Array (FPGA) of 75 MHz clock rate, and implementation on an 800 MHz ASIC (Application Specific Integrated Circuit) is going on. Vishkin et al. [15] claim that on a test set of 8 progams (such as binary tree search, vector compaction, matrix multiplication, or convolution), Paraleap is 1.9 to 9 times faster than a 2.6 GHz AMD Opteron processor. They expect that in near future the number of TCU's on a chip could grow to 1024.

3.2 Eclipse

In an on-going Finnish effort [4], a full architectural realization (so called Eclipse framework) of a strong PRAM model on a distributed memory chip multiprocessor (CMP) is being investigated. The main idea has been to provide enough communication bandwidth to be able to solve the routing problem described in Section 2 with a high probability, to use interleaved multithreading to hide the latency of the distributed memory system, and to synchronize the execution of emulated PRAM steps by an efficient wave synchronization technique. The communication solutions applied so far have been output buffered acyclic sparse mesh and multi mesh networks. See Figure 3.2.

Additional performance boost is sought by integrating instruction-level parallelism seamlessly with multithreading by chaining of functional units, which allows exploitation of virtual instruction-level parallelism even if executed threads are strictly sequential. Concurrent memory access to a single memory location is implemented with step caches that reduce the number of references per location to one per processor core. The data of pending multioperations are kept in the scratchpads, and active memory units are used to implement multiprefix operations [5].

According to recent investigations [7], the silicon area and power consumption

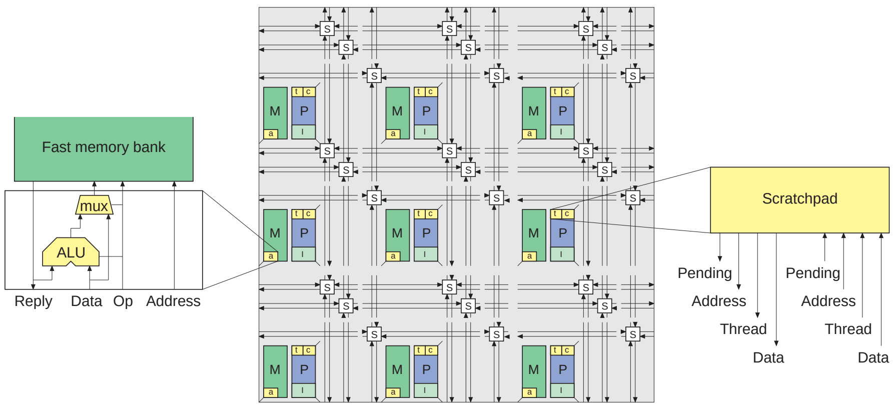

Figure 3: High-level block diagram of the Eclipse architecture (P=processor, M=data memory, I=instruction memory, a=active memory, c=step cache, and t=scratchpad memory).

of such a CMP is roughly comparable to those of contemporary multi-core offerings with the same number of cores from Intel. Interestingly also the performance/area and performance/power figures of the strongest PRAM variants have turned out to be the best. CMP has been investigated by software simulations. Implementation on FPGA is planned.

3.3 Moving threads

MOTH project studies the realization of a new kind of approach for mapping the computing of an application to MP-SOC architectures [6] (some preliminary ideas appear in [10, 11]). Instead of moving data read and write requests, we move extremely lightweight threads between the processor cores. Each processor core is coupled with memory module and parts of each memory module together form a virtual shared memory abstraction. Applications are written using a high-level language based on shared memory. As a consequence of moving threads instead of data we avoid all kinds of cache coherence problems.

In our architecture, the challenge of having efficient implementation of an application reduces to mapping the used data so that the need to move threads is balanced with respect to the bandwidth of the communication lines. This method also eliminates the need for separate reply network and introduces a natural way to exploit locality without sacrificing the synchronicity of the PRAM model.

In the moving threads approach, a multicore system consists of P processor cores that are connected to each other with some sparse network [10], e.g. with a butterly, a sparse mesh, a mesh of tree, etc. In traditional approaches, the messages correspond to read or write requests and replies, whereas in the moving threads approach, a message moves a thread consisting of a program counter, an id number, and a small set of registers. The messages in the moving threads approach are a little bit longer, but respectively there is no need for a network deliver the replies of read requests.

A cache-based access to the memory system is provided via each processor core. However, each core sees only a unique fraction of the overall memory space, and thus there are no cache coherence problems and when a thread makes a reference out of the scope of the core's memory area, the referencing thread must be moved to the core that can access that part of the main memory. Besides a cache to access the data memory, each core also has another cache for program instructions.

Each of the cores has $\Theta(s)$ threads to execute, and the threads are independent of each other – i.e. the core can take any of them and advance its execution. By taking an instruction cyclically from each thread, the core can wait for memory access taking a long time (and even tolerate the delays caused by moving the threads). The key to hide the memory (as well as network and other) delayes is that the average number of threads s per core must be higher than the expected delay of executing a single instruction from any thread.

The less there is need to move the threads, the smaller can the slackness factor be. Thus, although the moving threads approach does not require it, it might be wise to allow careful design of the allocation of actual data used in the program, and thus allow the programmer to balance the work-loads and to minimize the movement of data. We can e.g. assume that the program's address space is statically distributed into the memories accessible via cache modules attached to each core. The advantage of this is that the programmer can have influence on the physical allocation of data – and consequently on the physical allocation of the work of each thread on the processor-storage modules.

For the creation and termination of threads in the programming language level, we take the approach of supporting only implicit termination as well as creation of threads. We do not consider Java-like explicit declaration of threads as first-class objects as a feasible solution. In practice, we have a parallel loop-like construction which creates threads with logical id-numbers in the interval [low,high] and each threads is running the same program block. The code in the program block can of course depend on the logical processor id-number. The id-numbers are program controlled, but the runtime system expects them to be unique at anytime during the program execution. We also consider supporting nested thread creations. Each thread faces an implict termination at the end of the program block (which was defined in the thread creation statement).

4 Conclusions

In this section we discuss, what is common and what is different in the presented three parallel processor designs. The first difference is in the degree of existence and experience. Paraleap is moving from FPGA stage to ASIC stage and some programming experience has already been collected. Eclipse has not yet reached the hardware stage, and moving threads design is still quite sketchy. They all share the view that processor must support parallel processing, otherwise the latency gap prevents efficiency.

Paraleap requires long, independent parallel threads to run efficiently. Programmer must be aware of that and therefore programming is more difficult than just writing PRAM algorithms. Eclipse and moving threads processor are synchronous and allow more fine-grained parallel processing, which is easier for the programmer. This is theoretical reasoning. Currently the difficulty of Paraleap programming is dominated by undeveloped programming environment, which is less developed for Eclipse and nonexistent for moving threads.

In all designs, a part of the chip surface is dedicated for the internal network. Paraleap uses a mesh of trees for this purpose, while we believe in sparse mesh networks, due to scalability. There is no proof yet, which is better for this purpose.

In sequencial programming caches are a central tool to fight against latency, while cache coherence is an unsolvable problem. In parallel processing caches are just one useful trick. In Paraleap and in Eclipse, caches are between the internal network and the memory. Due to slackness, coherence is not a problem. In moving threads processor, cache must be next to each core, but due to them moving thread, coherence is not a problem.

Each of the designs have their own pluses and minuses. In Paraleap the prefix sum unit has a very central role in in thread scheduling. It is sequential and said to be fast enough. Is it scalable? Eclipse is apt for instruction level optimization and it provides multioperations used in some PRAM models. But how will it work as hardware? An interesting property of moving threads processor is that it needs only one-way traffic, but packets containing register environment are bigger. How do they balance?

A lot of research is needed to clarify these design questions. Parallel processor design is taking its first steps, while there is an enormous investment in sequencial processing. A funny proof of the need of rethinking is the CUDA project [3]: A graphics processor, when used as a parallel processor, is much more efficient than the main processor of the computer. What if we designed parallel processors for computing?

References

[1] P. Bach, M. Braun, A. Formella, J. Friedrich, T. Grun, C. Lichtenau. “Building the 4 processor SB-PRAM prototype.” In *Proc. of the 30th Hawaii International Conference on System Sciences: Advanced Technology Track - Vol. 5*, 1997.

[2] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subramonian, T. von Eicken. “LogP: Towards a realistic model of parallel computation.” In *Principles Practice of Parallel Programming, pp 1-12, 1993.*

[3] `http://www.nvidia.com/cuda`

[4] M. Forsell. “A Scalable High-Performance Computing Solution for Network on Chips.” *IEEE Micro 22(5)* (September-October 2002), pp. 46-55.

[5] M. Forsell. “Realizing Multioperations for Step Cached MP-SOCs.” In *Proc. SOC’06*, November 14-16, 2006, Tampere, pp. 77-82.

[6] M. Forsell and V. Leppänen. “Moving Threads: A Non-Conventional Approach for Mapping Computation to MP-SOC.” In *Proceedings of the 2007 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA’07)*, pages 232-238, Jun 2007.

[7] M. Forsell, J. Roivainen. "Performance, Area and Power Trade-Offs in Mesh-based Emulated Shared Memory CMP Architectures." In *Proc. PDPTA'08*, Jul 14-17, 2008.

[8] S. Fortune, J. Wyllie. "Parallelism in Random Access Machines". In *Proc. 10th ACM Symposium on Theory of Computing*, pp 114-118, 1978.

[9] J. J'aj'a. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

[10] V. Leppänen: *Studies on the Realization of PRAM*, PhD thesis, University of Turku, Department of Computer Science, TUCS Dissertation 3, November, 1996.

[11] V. Leppänen. "Balanced PRAM Simulations via Moving Threads and Hashing." *Journal of Universal Computer Science*, 4:8, 675–689, 1998.

[12] V. Leppänen, M. Penttonen. *Sparse Optical Torus*. AMICT'07 proceedings.

[13] G.E. Moore. "Cramming more components onto integrated circuits". *Electronics* 38(8), 1965.

[14] L.G. Valiant. "General Purpose Parallel Architectures". In *Handbook of Theoretical Computer Science, Ed. Jan van Leeuwen, Elsevier and MIT Press*, volume 1. Elsevier Science, 1990.

[15] X. Wen, U. Vishkin. "FPGA-based prototype of an PRAM-On-Chip processor." *Computer Frontiers 2008*, May 5-7, 2008.