International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 3, Number 1, pages 98–115, January 2013

An Extended PRAM-NUMA Model of Computation for TCF Programming

Martti Forsell
Platform Architectures Team
VTT Technical Research Centre of Finland
Box 1100, FI-90571 Oulu, Finland
Martti.Forsell@VTT.Fi

and

Ville Leppänen
Department of Information Technology
University of Turku
Joukahaisenkatu 3-5, FI-20014 Turku, Finland
Ville.Leppanen@UTU.Fi

Received: July 30, 2012
Revised: October 26, 2012
Accepted: December 4, 2012
Communicated by Akihiro Fujiwara

Abstract

The main problems with current multicore architectures are that they are difficult to program due to the asynchrony of the underlying model of computation and that the performance is weak with many parallel workloads due to architectural limitations. To address these problems we have introduced the Parallel Random Access Machine - Non Uniform Memory Access (PRAM-NUMA) model of computation that can be used to implement efficient shared memory computers for general purpose parallel applications with enough parallelism and yet support sequential and NUMA legacy code and avoid loss of performance in applications with low parallelism. While programming of computers making use of the PRAM-NUMA model is provably easy, there is still room for improvement since they make implementing time-shared multitasking expensive, sometimes replicate much of the execution unnecessarily, and force the programmer to use looping and conditional control primitives in the case the application parallelism does not match the hardware parallelism. Thick Control Flow (TCF) is a parallel programming model that does not provide a fixed number of threads like PRAM-NUMA but a number of control flows that have certain thickness that can vary according to needs of the application catching the best parts of the dynamicism and generality of the original unbounded PRAM model and simplicity of the Single Instruction Stream Multiple Data Streams (SIMD) model. In this paper we study the possibility to implement the TCF model on top of the PRAM-NUMA model and propose an extended PRAM-NUMA model that makes this straightforward. A number of variants of the extended model are identified and tied to existing execution models. Architectural implementation techniques and programming of the extended model and its variants are outlined and discussed with short examples.

Keywords: Parallel computing, Models of computation, Programming model, PRAM, NUMA, Thick control flow

1 Introduction

Even though times of exponential performance growth of sequential computers driven by rapidly increasing clock frequency are over, processor manufacturers are projecting that the performance growth could still continue – now driven by the increase of transistors per chip predicted by the original Moore's law [21] and consensus of a number of scientists and industrial specialists [1]. The idea is that the increased number of transistors make is possible to increase the number of processing elements per silicon platform that would by its turn realize as exponential speedup. The results have been modest so far – the potentially increased computational power has recently realized as performance growth in some applications while others have seen a little or no speedup with respect to sequential execution while the programming has become tedious compared to sequential counterparts even though the computational problems behind the applications are very often parallel in nature.

These problems – ease of programming and applicability to general purpose functionalities, i.e. high performance with a wide spectrum of workloads, are among the hardest challenges in parallel and multicore computing [28, 24, 13]. The roots of these challenges lay in the models of computation and architectures realizing them. Namely, the computational models employed in current machines are based on asynchronous threads blurring the exact state of the threads from a programmer and forcing him to use tricky asynchronous algorithms. In theory, one could easily use explicit thread-wise synchronizations even with current architectures [12] but in practice they are very expensive – each synchronization taking from hundreds to thousands of clock cycles – so that one needs to minimize the amount of them requiring very careful analysis of inter-dependencies between threads. Another problem is that the interconnection networks between the processor cores and memory modules are designed based on an assumption that most of the traffic is local and therefore do not typically have enough bandwidth to support random traffic without severe congestion. This together with the high cost of synchronizations tends to make impractical the algorithms that are applicable to current machines coarse-grained and thus limits the usage of current architectures in general purpose parallel computing.

In the history of parallel computing there has been attempts to provide direct support for easier-to-program synchronous models of computation and architectures that would provide enough bandwidth and therefore would allow for usage of fine-grained parallel algorithms and thus be more general purpose machines. One of the most ambitious and promising attempts has been the introduction of the *Parallel Random Access Machine* (PRAM) model [14] that extends the model of sequential computing by providing a number of processors that are connected to a shared memory and operate synchronously. While early implementation attempts [27, 15, 25] left much room for improvements in performance, implementability and scalability, the more recent ones [2, 17, 5, 29, 9] are much more sophisticated providing many highly efficient techniques to address the synchronicity and bandwidth constraints of current machines. Most of these make use of the *Emulated Shared Memory* (ESM) architecture, which consists of P processors, each T_p-threaded, that are connected to M (often equal to P) memory modules via a high-bandwidth communication network [26, 19] (see Figure 1). The idea behind the emulation is to use physically distributed memory but to provide a programmer an illusion of the PRAM-like unified shared memory and synchronous execution with efficient synchronization techniques.

Due to usage of multithreading, these architectures often have a utilization problem with workloads having low thread-level parallelism. To address this and to support execution of NUMA legacy code efficiently we have introduced the *Parallel Random Access Machine – Non Uniform Memory Access* (PRAM-NUMA) model of computation in which groups of threads belonging to the same processor can be set to mimic single threads of NUMA execution [10]. With an architecture implementing this model one can execute efficiently code independently of available parallelism [9].

Programming of PRAM-based parallel computers happens typically with a high-level programming language such as *Fork* [18] or *e* [7] that make use of a fixed number of threads as defined by the underlying *Multiple Input Stream Multiple Data Stream* (MIMD) ESM architecture. The threads are orchestrated by parallel control constructs that can command all the threads as a group or individual threads with thread-private conditions. While this kind of a programming model can

[0]This work was funded by the REPLICA project of VTT.

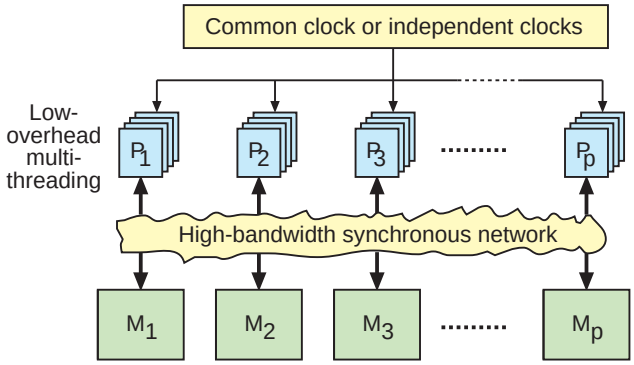

Figure 1: ESM architecture (P=multithreaded processor, M=memory module).

provably be used to implement a rich granularity-independent parallel algorithmics [16, 17], its fixed set of threads forces the programmer to use looping, possibly temporary variables and conditional control primitives in the case the application parallelism does not match the hardware parallelism, sometimes replicates much of the execution unnecessarily, and makes implementation of multitasking costly since there are multiple contexts to be switched per processor rather than only one in single threaded processors.

In our earlier work we have introduced the *Thick Control Flow* (TCF) parallel programming model that does not provide a fixed number of threads like PRAM-NUMA but a number of control flows that have certain thickness that can vary according to the needs of application catching the best parts of the dynamicism and generality of the original unbounded PRAM model and simplicity of the *Single Instruction Stream Multiple Data Streams* (SIMD) model [20]. In this paper we study the possiblities to implement the TCF model on top of the PRAM-NUMA model and propose an extended PRAM-NUMA model that makes this straightforward. A number of variants of the extended model are identified and tied to existing execution models. Architectural implementation techniques and programming of the extended model and its variants are outlined and discussed with short examples.

The rest of the paper is organized so that in Section 2 we describe the PRAM-NUMA model of computation and TCF programming model. In Section 3 we extend the PRAM-NUMA model to support TCF, identify a number of variants to the model, and discuss the architectural realization of it. In Section 4 we consider TCF-aware programming with simple examples. Finally, in Section 5 we give our conclusions.

2 PRAM-NUMA model of computation and TCF programming model

In order to understand the locality and low parallelism driven problems in ESM-based PRAM realizations, we take a look at our recent model providing fast operation in low-TLP cases with NUMA capabilities while retaining power of the PRAM model. Unlike original PRAM, this model is bounded and provided with the concept of distance metric, modeling the relative distance of the processors. We introduce also the TCF programming model that promises to simplify high-level programming of parallel computers by dropping the concept of threads especially as its fixed number incarnation.

2.1 PRAM-NUMA

The *PRAM-NUMA model of computation* is a configurable synchronous shared memory model [10]. It consists of T processors grouped as P groups of T_p processors, a word-wise accessible global shared memory, P local memory blocks, a metric defining distance between the processor groups and target memory blocks, and distance-aware interconnection network (see Figure 2). Each processor is attached to the shared memory and each processor group is attached to its own local memory block. The interconnection network connects the local memory access paths of processor groups together and is distance-aware in a sense that the latency of routing is proportional to the distance between the source processor and destination memory block. The bandwidth of a group of processors to the shared memory and local memory are the same. Each processor can be configured to either PRAM mode or NUMA mode. Along with configuration from the PRAM mode to the NUMA mode, one can set the state of the processor to point an arbitrary state within the group it belongs to. With this indirection of states, two or more processors belonging to a group can be configured to a NUMA bunch so that they execute a common instruction stream and share their state with each other, i.e. execute code like a single processor.

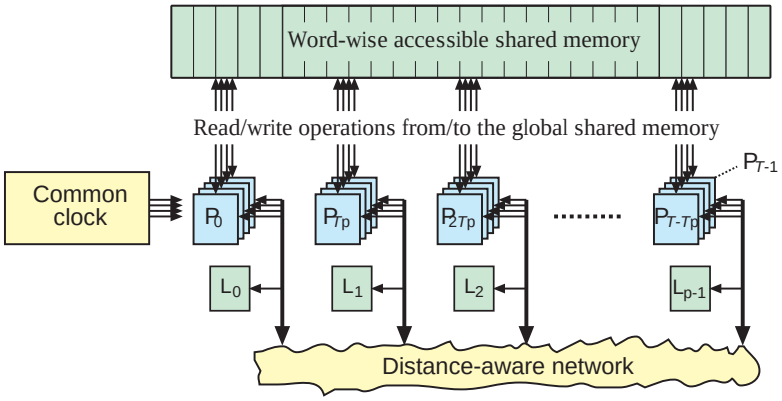

Figure 2: PRAM-NUMA model (P=processor, L=local memory).

Execution in the PRAM-NUMA model happens in synchronous steps during which every processor of each group executes exactly one instruction in the PRAM mode and every bunch executes exactly as many instructions for a single instruction stream as there are participating threads in the bunch.

The PRAM-NUMA model solves the low-TLP execution problem of ESMs by providing a possibility to configure two or more processors of a group to use a common state like they were a single processor and to perform NUMA access to the local memories. In the case of low TLP portion of code, a programmer can just set up to T_p processors per group to run that portion as a NUMA bunch and gain more performance proportionally to the number of processors in the bunch.

2.2 Thick Control Flow

We have developed *Thick Control Flow* (TCF) as a parallel programming model [20] that does not provide a fixed number of threads like PRAM-NUMA but a dynamically varying number of control flows that have certain thickness that can vary according to needs of the application catching the best parts of the dynamicism and generality of the original unbounded PRAM model and simplicity of the SIMD model where applicable. TCF computation can be characterized with a block structure shown in Figure 3. The control flow between blocks is illustrated with thick arrows defining sequential execution relations between the blocks. Some of the blocks can be executed in parallel and the execution within blocks is synchronous. Each block has a thickness whose idea will be explained next.

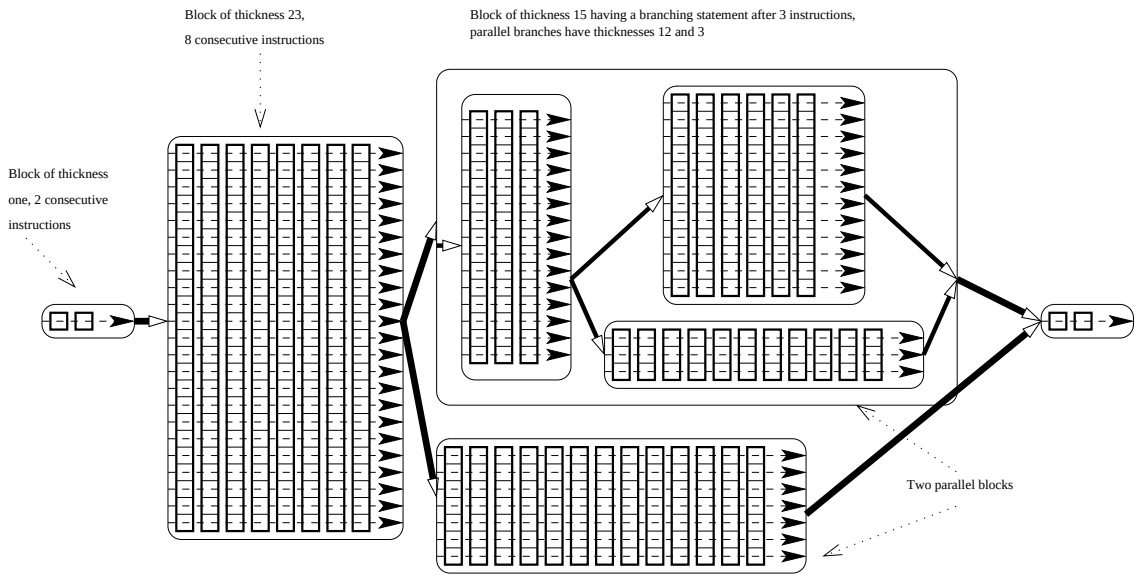

Figure 3: An example of executing functionality with TCFs.

When a thick control flow (in terms of the number of implicit threads) is executing a statement or an expression of a program, all the threads of the flow are considered to execute the same program element synchronously in parallel. As all the implicit threads of a flow have a unique identity (number), each executed "same" instruction can have identity related parameters meaning that the effect is different for each implicit thread.

The unconventional semantics of method calls for TCF is claimed novel. When a control flow with thickness T calls a method, the method is not called separately by each of the T threads, but the control flow calls it only once with T threads. A call stack is not related to each thread but to each of the parallel control flows, since threads do not have program counters – only control flows have program counters. In fact, the concept of thread is only implicit. A thick thread-wise variable is an array-like construct having a thread-wise actual value. Method signatures naturally advance types with thickness, but non-thick types are also useful.

Originally, a program is considered to have a flow of thickness one, measured conceptually in number of parallel implicit threads. A method can be considered to have a thickness related to the calling flow's thickness. For dynamically changing the thickness of the flow, we have *thick block statement*, which sets a new thickness for a block or a *thickness statement* that sets the thickness for the statements to be executed. For the former, nesting thick and ordinary block statements is supported. Consider a situation where a thick block B_{outer} of thickness T_{outer} contains an inner thick block B_{inner} of thickness T_{inner}. A nested block is not executed thread-wise but flow-wise, and therefore considering the flow thickness, a flow executing the inner thick block has thickness T_{inner} (instead of $T_{outer} \times T_{inner}$). In the case of statement setting the thickness for the statements to be executed, the thickness for each code segment is specified explicitly.

Executing a control statement (if, switch, ...) can temporarily mean splitting a thick control flow into several other flows as illustrated in Figure 3. The resulting potentially non-continuous thread subgrouping (non-continuous indexing of implicit threads) is considered rather costly to implement. Thus, in this paper each parallel branch is considered as a nested thick block with thickness determined by the number of implicit threads "selecting" the branch. The implicit threads of the surrounding block will not continue in the blocks determined by the branching statement. As the above is equal to parallel execution of multiple paths with given thicknesses, we require in this paper that the whole flow selects exactly one path through a control statement. If a programmer wants to execute multiple paths in parallel, he should give a parallel statement creating multiple control flows accordingly, and set the thicknesses for them. Besides splitting the current flow into a

number of parallel flows, a parallel statement also performs an implicit join of the flows back to the calling flow at the end of the statement. All threads of a control flow can be seen to synchronously march through the common program code like in a dynamic SIMD model. When a flow is split into separate flows, nothing can be assumed about the advancing speed of the split flows – i.e., in this sense the parallel flows are asynchronous with respect to each others. However, if the programming language designer wants to, he can make execution synchronous at machine instruction level by applying the correct variant of the extended PRAM-NUMA model (see Section 3).

The concept of thick control flow makes the programmer to focus on co-operation of few parallel thick control flows instead of a huge number of parallel threads. The concept of computation's state is promoted as a flow is seen to have a state (instead of each thread) and parallel operations within each thick flow take place synchronously. As the concept of state has been in a central role in achieving correctness in sequential programs, we believe such constructions very desirable that support the visibility of (abstract) computation's state for the programmer.

For the rest of the article we use the visualization style shown in Figure 4. In it, instructions of a TCF are shown as stacks of thread-wise operations. As the height of the stack alters so does the thickness of the TCF. Note also that for the computational model variants in Section 3.2, execution of TCFs in extended PRAM-NUMA-aware ESM architecture is shown in a single processor view in which the TCF and bunch slices are executed one-by-one sequentially for latency hiding reasons (see Figure 6).

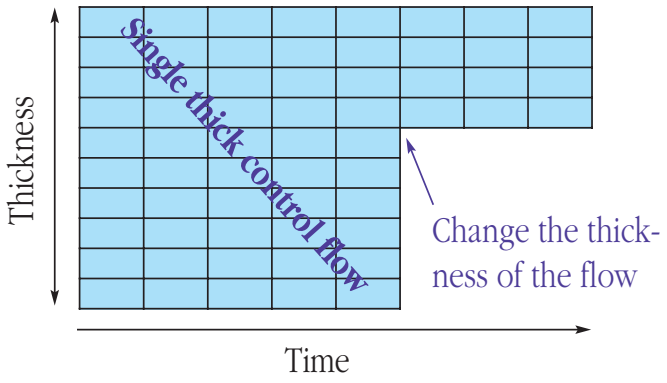

Figure 4: Execution of a TCF that changes thickness.

2.3 TCF in contrast to contemporary SW / HW models

TCF resembles a bit the model used in the XMT architecture [29] but XMT executes threads asynchronously from their creation to the termination dropping the lock-step synchronicity of the PRAM model while TCF executes instructions in a similar way and retains the synchronicity of PRAM. TCF also resembles so-called stream computing model of GPGPU devices, e.g. the Kepler [22]. In stream computing the focus is on data streams which are operated with kernel functions – the computing flow is essentially sequential in stream computing and typical stream programs do not have any nested call-structure between the kernels. TCF applies also to object-oriented computing and it naturally generalizes the ordinary sequential computing based on modules and rather deeply nested call-structures.

There are no architectures making use of the TCF programming model as such, but some essential techniques like throughput computing and certain weaker variants of TCF are supported in existing machines and academic architectures. The SIMD model uses the same data parallelism idea as TCF but it has only one control flow whose thickness is not flexible.

Exposing the very different details of the underlying machine via programming model (as qualifiers or attributes) and making the programmer to be aware of the machine properties, is a popular approach which turns the programmers as optimizers and designers of physical placement and temporal usage related to locality of program data. As optimization using HW details is challenging,

these approaches are one primary reason for parallel programming considered difficult. E.g. Cuda, OpenCL and OpenMP of heterogeneous parallel programming [3] are good examples. We believe that machine details should be abstracted if possible (as pointed out in (the report version of) [4]) to value programmer's productivity [4, 3].

One more approach is to have parallel control structures in the language. E.g. kernels of Cuda / OpenMP (stream computing) can be seen as functions capsulating the body of a parallel loop. The FORALL construct of High Performance Fortran was in the same role. Languages like Fork and Replica have parallel loops but – unlike in TCF – they restrict function calls (the base of modularity) to apply only thread-wisely (Cuda's new dynamic programming does not suffer from this). The thick flow of TCF has clear similarities with kernels of stream computing. Even the structured threading of X10 (a PGAS language of High Productivity Computing project) resembles the thick flow concept. Moreover, from task parallelism point of view, the TCF can be seen so that tasks map with parallel thick flows and within tasks GPGPU-related SIMD-style of execution is applied.

3 Extending the PRAM-NUMA model

The PRAM-NUMA model of computation is not directly compatible with the TCF programming model, since TCFs of variable thickness can not be directly mapped to a fixed number of threads. In order to provide support for dynamic and flexible TCF programming we extend PRAM-NUMA by replacing the concept of thread by thick control flow. We define here the model and its variants, and discuss possible architectural implementations.

3.1 Model

The *extended PRAM-NUMA model of computation* is a configurable synchronous shared memory model. It consists of T TCF processors grouped as P groups of T_p processors, a word-wise accessible global shared memory, P local memory blocks, a metric defining distance between the processor groups and target memory blocks, and distance-aware interconnection network (see Figure 5). Each TCF processor is attached to the shared memory and each processor group is attached to its own local memory block. The interconnection network connects the local memory access paths of processor groups together and is distance-aware in a sense that the latency of routing is proportional to the distance between the source processor and destination memory block. The bandwidth of a group of processors to the shared memory and local memory are the same. Each TCF processor can be configured to either PRAM mode or NUMA mode. Along with configuration, one can set the thickness T of TCF execution within the group it belongs to. With this thickness setting a TCF processor can be configured to execute either T identical (SIMD-style data parallel) operations or T consecutive instructions (thickness $\frac{1}{T}$) so that variable length vectors as well as low-parallelism blocks can be processed without looping and/or overhead. If the thickness is set to zero then the processor does not execute anything.

Execution in the extended PRAM-NUMA model happens in *synchronous steps* during which every processor of every group executes exactly one TCF instruction consisting of T identical operations (resembling interleaved data parallel or SIMD instruction execution) in the PRAM mode or T consecutive instructions in the NUMA mode.

3.2 Variants

By altering the execution scheme and order we get the following variants of the extended PRAM-NUMA model indicating the generality of it and promoting the theory of parallel models of computation. The illustration of the execution assumes that these models make use of the ESM-like latency hiding scheme where applicable and NUMA elsewhere. Thus, the processors belonging to a group are emulated by a multithreaded processor in the PRAM mode and by the bunched processor in the NUMA mode (see Figure 6).

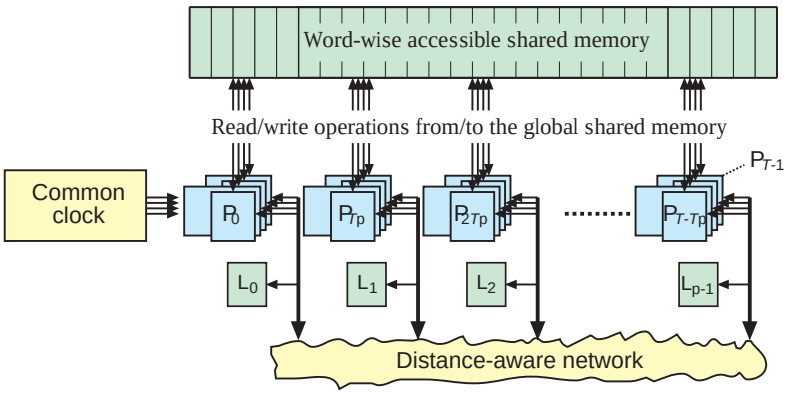

Figure 5: Extended PRAM-NUMA model (P=TCF processor, L=local memory).

Single instruction: Execution in the extended PRAM-NUMA model happens in steps during which every processor of each group executes exactly one TCF instruction consisting of a variable number of identical operations in the PRAM mode or a variable number of instructions in the NUMA mode (see Figure 7). This is the most general variant, realizing the TCF programming model to full extent. However, if multiple flows with different thickness settings are allocated for a single TCF processor group, this variant can lead to unbalanced execution in which thick instructions slow down the execution of thin instructions in efficiency sense. Efficient employment of all TCF processors would then require splitting overly thick instructions to thinner ones and thus allocating the execution of a single flow to several TCF processors (see the Balanced variant).

Balanced: Execution in the extended PRAM-NUMA model happens in steps during which every processor of each group executes a bounded number of operations out of TCF instructions in the PRAM mode or a bounded number of instructions in the NUMA mode (see Figure 8). In the case a TCF instruction is not completed in a single step, execution is continued from the first unexecuted operation/instruction until the bound interrupts the execution again or all the operations are executed. This does not effect the programmability of the model, but just the scheduling of instructions.

Multi-instruction: Execute multiple instructions in a logical TCF step. Note that despite of allowing multiple instructions per step it is still possible to use multiple threads for latency hiding. If the NUMA support is dropped and execution starts from the creation of the TCF and continues till the termination, we get the execution model used in the XMT architecture [29] (see Figure 9). The advantage of this is that execution of parallel constructs becomes simple and flexible unless they contain inter-dependencies. By doing this we, however, lose the synchronicity of the PRAM model and coarsen the granularity of computation.

Single-operation: Set the thickness of all TCFs to one. This brings back the utilization problems in the low-parallelism case and reintroduces the thread arithmetics and replication problems. If we now drop the NUMA support we obtain the standard interleaved ESM architecture used in PRAM realization attempts, e.g. ECLIPSE [5] and SB-PRAM [17] (see Figure 10).

Configurable single operation: Set the thickness of all TCFs to one and allow bunching of processors. This leads back to the original PRAM-NUMA model implemented e.g. in TOTAL ECLIPSE architecture [9] (see Figure 11). Thus, the utilization problem in the low-parallelism case is eliminated but the thread arithmetics problem stays.

Fixed thickness: Set the thickness T_p of TCFs to a fixed value, drop the NUMA mode, and add a scalar processing unit. If we at the same time also limit the number of processors to one, we

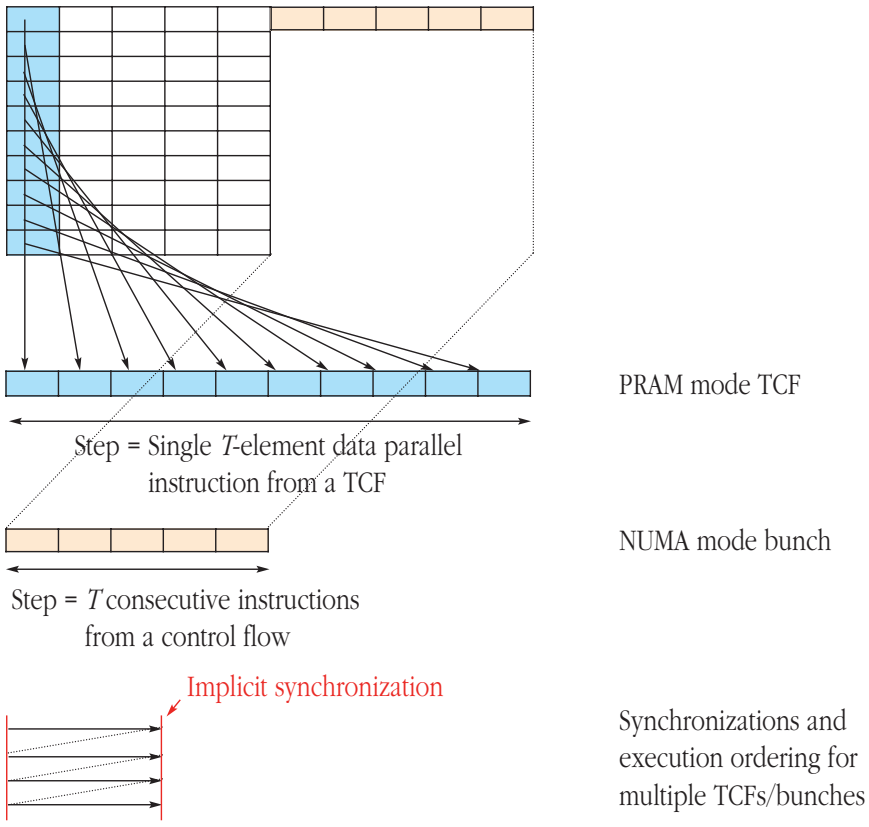

Figure 6: Execution of TCFs and NUMA bunches on an ESM-style architecture (single processor group view).

get the traditional vector/SIMD model (see Figure 12). By doing this we make it impossible to execute control parallel algorithms efficiently. Also the thread arithmetics problem and likely the utilization problem are reintroduced.

At this point some readers might be tempted to ask whether selecting data flows instead of control flows would have given similar results. Without trying to prove it here, it is obvious that besides inducing a way different theory, a data flow-based model would also make most general purpose algorithms much harder to construct since in our understanding the data flow model tends to distribute control in a highly non-trivial way into the computation.

Another direction of interest might be heterogeneity either in control flows or in processor groups. While our definition of TCF requires that the operations applied to the elements of TCF are homogeneous, it is possible and even advisable to apply heterogeneous instruction-level parallelism to execution of TCFs. In our previous work we have shown how ILP-TLP co-execution can be very efficiently applied to ESM and CESM architectures implementing the PRAM and PRAM-NUMA models, respectively [6, 9]. Applying ILP without any TLP leads back to problems of limited and hard-to-extract instruction-level parallelism, weakly scalable superscalar/VLIW architectures, and compiler complexity problems that actually made the processor manufacturers to switch to multicore architectures. Finally, heterogeneous processor groups inevitably introduce various mapping, partitioning, locality problems that make programming and execution of general purpose functionality painful and potentially inefficient.

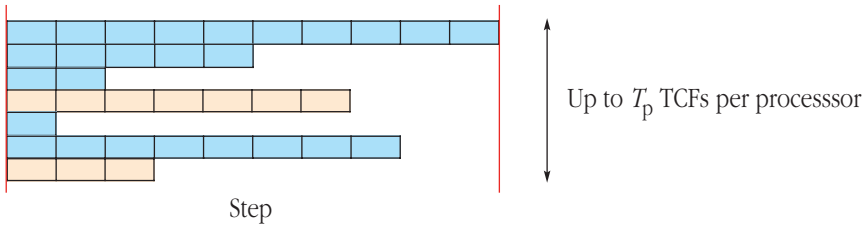

Figure 7: Execution on the single-instruction variant (single processor group view).

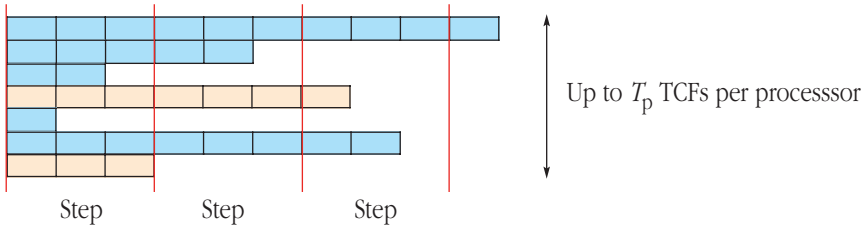

Figure 8: Execution on the balanced variant (single processor group view).

3.3 Implementation considerations

Although there are no known architecture for the true TCF-aware PRAM-NUMA (single instruction and balanced) variants, there exists physically feasible hardware implementation architectures for the multi-instruction variant [29], single-operation variant [5], configurable single operation variant [9], and especially for the fixed thickness variant of the extended model. In those existing implementations, the TCF model need to be emulated with software, and therefore the focus of our implementation considerations is on the true TCF-aware PRAM-NUMA variants. Since the extended PRAM-NUMA model resembles to that of the original PRAM-NUMA, we decided to advance the PRAM-NUMA implementation architecture *Configurable Emulated Shared Memory Machine* (CESM) [9, 10] as a starting point for our implementation considerations.

In order to support TCFs instead of threads there needs to be T_p-element storage block, e.g. ring buffer or addressable register file that contains the TCF information, e.g. thickness and mode as well as a pointer to the next yet not executed operation in the case of the balanced variant. Figure 13 shows an outline of the CESM employed with a TCF storage buffer attached to instruction fetch and operand select stages. Since the thickness of TCFs and length of bunches is not constant, operation of the execution pipeline can not be the same as in CESM making use of T_p pipeline stages each holding a thread. Instead, we could execute TCFs one by one making use of the pipeline in the repetitive way: Execution is started by fetching the next nonempty TCF from the TCF storage block. As the

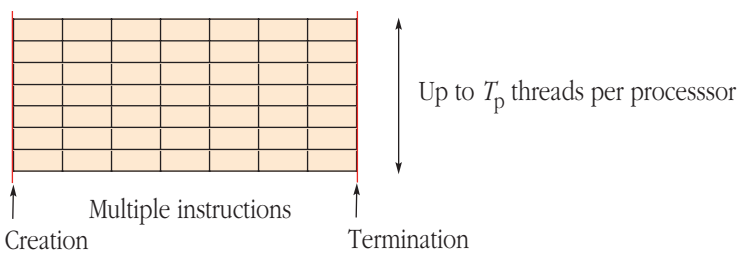

Figure 9: Execution on the multi-instruction variant assuming the execution continues from creation of threads to their termination (single processor group view).

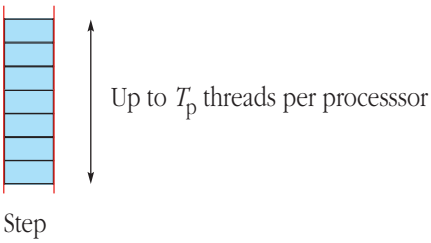

Figure 10: Execution on the single-operation variant (single processor group view).

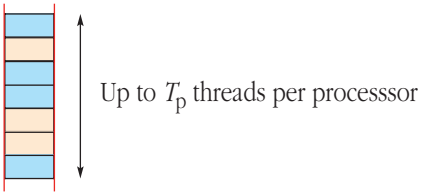

Figure 11: Execution on the configurable single operation variant (single processor group view; blue refers to PRAM operations and orange refers to NUMA mode operations).

TCF in execution turn enters into execution, the related instruction needs to be fetched (IF stage) and the common operands need to be selected just like in the CESM (OS stage). After that the TCF instruction needs to be halt in the pipeline and generate data parallel operations with unique identifiers according to the thickness setting of the TCF. As the final operation gets generated, the TCF needs to be released and advanced through the rest of the pipeline. The next TCF needs to be selected into execution as soon as there is room in the pipeline for it. If the thickness of the TCF is less than 1, say $1/T$, it is in the NUMA mode and there is a need to fetch T instructions from the instruction memory guided by the running PC value and execute each instruction in the beginning of the pipeline making use of the forwarding network, just like the CESM architecture does [9].

The most challenging aspect of implementing TCF-style execution is the in-principle unbounded thickness of TCF (and length of NUMA mode bunches). Since most parallel computations make use of some intermediate results, there is a need to store them somewhere. We see three possible solutions for this: memory-to-memory instructions, cached register file, and usage of a number of fast local memories. In memory-to-memory instructions, the thread-specific operands are located in the local and global memories like in the early days of computers. The cached register file allows one to maintain a high number of virtual registers with the help of a limited size register block acting as a cache to register accesses [23]. In the local memory based solution, the thread-specific operands are located in the local memories. See [11] for our early study on adding the local memory support also for the PRAM mode operation of the CESM architecture and note that similar amount of memory is needed for storing intermediate results if looping is used. Providing sufficient data access bandwidth to support multiple functional units can, however, be problematic especially in the local memory-based solution. On the other hand, this kind of TCF execution would considerably decrease the instruction memory bandwidth requirements in the PRAM mode assuming the thickness of TCFs is greater than one – one needs to fetch the instruction word only once per TCF. Unfortunately this is not true for the NUMA mode execution. In addition to reducing the required instruction

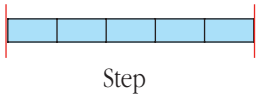

Figure 12: Execution on the fixed thickness variant (single processor group view).

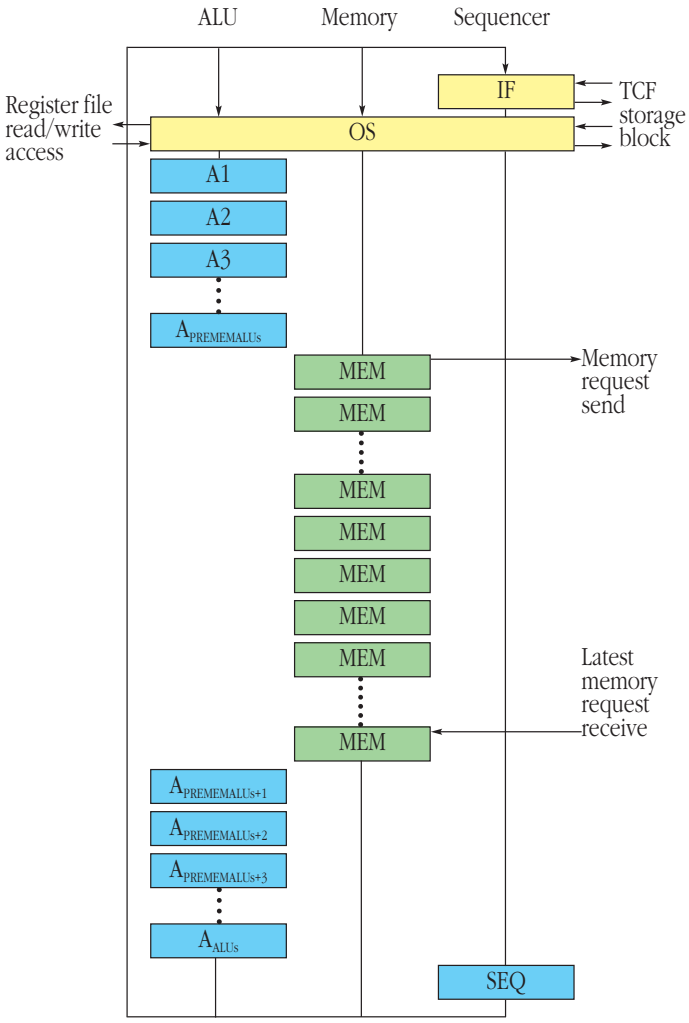

Figure 13: Simplified block diagram of the MBTAC processor with TCF storage block.

memory bandwidth, this kind of an architecture would drop the overhead of multitasking significantly since tasks can be considered as TCFs for which there would be efficient hardware assisted execution support anyway. Finally, the extended model would improve the utilization of data parallel execution by eliminating the need for replicating registers with identical value for all the threads. Table 1 summarizes the key properties and our cost estimations for various primitives for the variants of the extended PRAM-NUMA model.

Moving the focus from the TCF processor execution considerations to the overall TCF program execution reveals other challenges. When TCF instructions (TCF flows or TCF flow fragments) are allocated to TCF processors, for efficiency reasons it is necessary to try to keep the sum of thickness values at each TCF processor roughly balanced. Luckily, TCF computing provides two means for this. As in the TCF computing nothing is assumed of the relative execution speed of parallel flows, we can have an arbitrary subset of available parallel TCFs in execution at any moment. On the other hand, a flow is taken into execution as a whole, but its execution can be split to balanced fragments that are allocated to different TCF processors. Naturally, this kind of balanced single instruction variant based execution requires compiler and OS support. Observe that splitting an overly thick flow does not need to be done for each instruction separately, but the OS can split such flows automatically.

Table 1: Key properties and estimated cost of some primitive operations in the extended PRAM-NUMA variants (b=bounded variable, m=small constant, P=number of processor cores, R=number of register per thread or common registers for a TCF threads, T_p=threads per processor, u=unbound variable, limited by the size of the local memory and maxint).

	S-Instr.	**Balanced**	**M-Instr.**	**S-Op.**	**C. S-Op.**	**F-Thickn.**
Number of TCFs	$P \times T_p$	$P \times T_p$				
Number of threads	u	u	$P \times T_p$	$P \times T_p$	$P \times T_p$	$P \times T_p$
Registers per thread	$R/u + m$	$R/u + m$	R	R	R	R
Fetches per TCF	1	u/b	T_p	T_p	T_p	T_p
Cost of task switch	0	0	$O(1)$	$O(T_p)$	$O(T_p)$	$O(T_p)$
Cost of flow branch	$O(R)$	$O(R)$	$O(1)$	$O(1)$	$O(1)$	$O(1)$
PRAM operation	yes	yes	no	yes	yes	no
NUMA operation	yes	yes	yes	no	yes	no
Sequential operation	NUMA	NUMA	single thr.		NUMA	scalar unit
MIMD	yes	yes	yes	yes	yes	no

Architectural implementation at the machine level could likely be made in the same way as in CESM in which the machine consists of P MBTAC – processors that are connected into a distributed shared memory via a synchronous high-bandwidth communication network. This is because from the network point of view, TCF-enabled processors send memory references and synchronization messages just like MBTAC processors do in CESM.

4 Programming considerations

In our earlier work we have outlined a development methodology for ESM CMPs consisting of a strong computational model, a c-like TLP language *e*, ILP-TLP optimization algorithm, and application development flow [8]. It allows a developer to write parallel applications with a help of supporting theory of parallel algorithms [16, 17] and using parallel programming techniques from fully asynchronous coarse grained threads (processors in the PRAM-NUMA model terminology) down to synchronous threads interchanging information with the finest granularity. With this methodology, a computer with the extended PRAM-NUMA model can be programmed like any ESM machine for parallel enough functionality if the thickness of TCFs is set to one. By using thicknesses higher than one, a programmer can make use of simplicity and dynamism of the TCF programming model. This changes the way how some constructs are expressed since non-continuous thread numbering is not available any more. There are changes also in the usage of the NUMA mode since the extended PRAM-NUMA model uses a single TCF with thickness setting of $1/T_n$ to instantiate NUMA execution of T_n consecutive time slots while the original model combines $T_n \leq T_p$ threads freely selectable from the threads of a single processor into a NUMA bunch.

From the programming point of view some variants of the extended PRAM-NUMA model are treated a bit differently than others. The Single-operation variant naturally uses the convention of the PRAM model with fixed number of threads. The Configurable single operation variant adds a possibility to configure some threads to NUMA mode, which is typically denoted by the *numa* construct at the high-level language. The multi-instruction variant can not make use of the lock step synchrony of PRAM execution, thus it is typically programmed with *fork* constructs creating a number of asynchronous threads that are scheduled dynamically to the thread units [29]. Synchronization happens as the threads are joined back to a master thread. The Fixed thickness variant can not make use of control parallelism, i.e. there is no possibility to split the current execution into a number of parallel sub-executions. The Single instruction and Balanced variants are programmed in the same way. The only difference is that the execution rate between the different TCFs is mode balanced in the latter and that the number of steps is not any more connected to the number of executed instructions in the PRAM mode. This may help to keep the execution of

the processor more predictable in some real-time applications but will introduce some performance penalty due to more frequent synchronizations.

Assume that in the following we use the thickness statement that divides the given thickness evenly among all the processors rather than the thick block statement for setting the thickness of a TCFs. Since the PRAM-NUMA model (and thus the Configurable single operation variant) provides the programmer with a fixed number of threads, he must use a loop or a conditional statement to handle usual situations in which the problem size does not match the number of threads defined by the underlying machine:

```
for (i=_thread_id; i < size; i+=_number_of_threads)
    c_[i]=a_[i]+b_[i]; // More data elements than threads
```

or

```
if_ (_thread_id < size) // Less data elements than threads
    c_[_thread_id]=a_[_thread_id]+b_[_thread_id];
```

This works also with the Single operation variant. In the extended PRAM-NUMA model these can be replaced with just

```
c_=a_+b_; // Thickness matches with the size
```

that compiles to a non-looping sequence of native assembler instructions. The only precondition is to set the thickness of flow to size with

```
#size; // Set thickness to equal size
```

somewhere before the statement. In the Multi-instruction variant one does not need to specify a loop but a *fork* construct that creates a thread for each element of the arrays, i.e.

```
fork (_thread_id=0; _thread_id<size)
    c_[_thread_id] = a_[_thread_id] + b_[_thread_id];
```

This does not, however, work if there are dependencies between the threads.

If the number of data elements is too low for latency hiding in the PRAM-NUMA model, one needs to write

```
numa
    if (_processor_id<size) // Less data elements than procs
        c_[_processor_id] =a_[_processor_id]+b_[_processor_id];
```

which can be used in the Configurable single operation variant normally but drops the utilization of the processor to $1/T_p$ in the Single operation variant. In the extended PRAM-NUMA model, it is enough to write

```
c_=a_+b_;
```

assuming the compiler is able to detect the low parallelism situation and automatically launch the NUMA mode for the statement or alternatively the programmer declares NUMA execution explicitly by specifying block length T with

```
#1/T;
c_=a_+b_;
```

In order to avoid splitting the current TCF to non-continuously numbered TCFs like defined by one-way conditional statement

```
if_ (_thread_id < size/2)
    c_[_thread_id]=a_[_thread_id]+b_[_thread_id];
```

in extended PRAM-NUMA, we just write

```
#size/2;
c_=a_+b_;
```

while the two-way conditional construct defining statements to be executed in parallel

```
if_ (_thread_id < size/2)
    c_[_thread_id]=a_[_thread_id]+b_[_thread_id];
else
    c_[_thread_id]=0; // Clear the upper part of c_
```

becomes

```
parallel_ {
    #size/2: c_=a_+b_;
    #size/2: c_[#+_id]=0;
}
```

which creates two TCFs for the duration of the construct. For the Fixed thickness variant, this needs to be done sequentially due to lack of control parallelism, e.g.

```
if_ (_thread_id < size/2)
    c_[_thread_id]=a_[_thread_id]+b_[_thread_id];
if_ (_thread_id >= size/2)
    c_[_thread_id]=0;
```

in which the *if* construct compiles to conditional execution for the vector execution elements.
Multioperations and multiprefixes with non-idealities requiring looping in the PRAM-NUMA

```
for (i=_thread_id; i < size; i+=_number_of_threads)
    prefix(source_[i], MPADD, &sum_, source_[i]);
```

extend similarly to the new model

```
prefix(source_, MPADD, &sum_, source_);
```

Dependent loops in the original PRAM-NUMA model can be executed without explicit synchronizations

```
for_ (i=1; i < size; i<<=1)
    if (_thread_id - i >= 0)
        source_[_thread_id] *= source_[_thread_id-i];
```

Note that the *if* construct as well as the synchronization at the end can be dropped if there is a *size*-element array initialized to zero prior to *source_* array in the memory.

In the Multi-instruction variant, synchronizations provided by the *fork* construct are needed with the cost of remarkable overhead but there is no need for the *if* construct assuming the numbering of the threads can be started from *i*.

```
for (i=1; i < size; i<<=1)
    fork (_thread_id=i; _thread_id<size)
        source_[_thread_id] *= source_[_thread_id-i];
```

For the extended PRAM-NUMA model this can be written

```
for (#=size-1; #>0; #>>=1)
    source_[_id+size-#] *= source_[_id];
```

A possible drawback of the TCF model is that asynchronous programming convention making use of the possibility that threads are not in synchrony becomes more expensive by spending a TCF with thickness one for each asynchronous thread like in the original PRAM-NUMA model. On the other hand, specifying even close to $P \times T_p$ individually controlled threads is a huge and complex task.

Time-shared multitasking is expensive in ESM, CESM and the original PRAM-NUMA since it requires switching all the threads taking T_p times more time than that in a single threaded computers (see Table 1). In the extended model TCFs can be treated as tasks and since switching between TCFs is very cheap – it takes no time – as long as all the TCFs fit into the TCF storage block of the architecture. For performance and load balancing purposes it is much more beneficial to allocate horizontally $T_{application}/P$-wide TCFs from each processor core rather than to do it vertically, e.g. by allocating a single $T_{application}$-wide TCFs from a single processor core. Physical partitioning of the machine onto a number of sub-machines works also well if balanced correctly but is much more static than the horizontal TCF allocation.

Strictly sequential portions of code set challenges equally to PRAM-NUMA and its extended version since a bunch from only one processor can be utilized leaving the rest of the processors potentially idle. This is however true for all multicore processor architectures.

5 Conclusions

We have proposed an extended PRAM-NUMA model of computation to support the TCF programming model in ESM parallel computers by replacing the concept of thread with the concept of TCF. In our opinion, the TCF-aware model interestingly combines the best parts of the unbounded PRAM model and the simplicity of the SIMD model. With a TCF machine featuring fully configurable thickness one can easily avoid a lot of thread arithmetic and looping/conditional statements that are needed in non-ideal cases of the application dataset not matching with the architecture defined thread set. Other advantages include greatly simplified implementation of multitasking and improved utilization of data parallel execution. The former is because tasks can be considered as TCFs and the obvious implementations of the model support fast switching between TCFs, and the latter because the TCF model eliminates the need to use the MIMD model for SIMD computations inside processor groups. We were able to identify a number of variants to the extended model and tied them to existing execution model to show the generality of this approach. We discussed also about architectural implementation of the model and outlined some principles noting that there remains some unsolved challenges for further investigation. Finally we considered programming with simple examples and illustrated how statements and expressions could be simplified with a help of TCF programming style in cases current models require complex constructs.

Our future work includes attempts to create an execution architecture and compiler for the extended PRAM-NUMA model based on our original PRAM-NUMA and TCF efforts. We will also study possibilities for adopting some of the ideas of TCF to our new PRAM-NUMA architecture called REPLICA.

References

[1] International Technology Roadmap for Semiconductors, 2011. Semiconductor Industry Association, 2011, http://www.itrs.net/.

[2] R. Alverson, D. Callahan, D. Cummings, B. Kolblenz, A. Porterfield, and B. Smith. The Tera Computer System. In *Proceedings of the International Conference on Supercomputing*, pages 1–6. ACM, New York, 1990.

[3] J. Diaz, C. Munoz-Caro, and A. Nino. A Survey of Parallel Programming Models and Tools in the Multi and Many-Core Era. *IEEE Transactions on Parallel and Distributed Systems*, 32(8):1369–1386, 2012.

[4] K. Asanovic et al. A view of the parallel computing landscape. *Communication of the ACM*, 52(10):56–67, 2009.

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

[6] M. Forsell. Using Parallel Slackness for Extracting ILP from Sequential Threads. In *Proceedings of the SSGRR-2003s, International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet, July 28 - August 3, 2003, LAquila, Italy*, 2003.

[7] M. Forsell. E – A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs. *WSEAS Transactions on Computers*, 3(3 (July)):807–812, 2004.

[8] M. Forsell. Parallel Application Development Scheme for General Purpose NOCs. In *Proceedings of the 2005 ECTI International Conference (ECTI-CON), Pattaya, Thailand*, pages 819–822, 2005.

[9] M. Forsell. *TOTAL ECLIPSE – An Efficient Architectural Realization of the Parallel Random Access Machine*, pages 39–64. IN-TECH, Vienna, 2010. Editor: Alberto Ros.

[10] M. Forsell. A PRAM-NUMA Model of Computation for Addressing Low-TLP Workloads. *International Journal of Networking and Computing*, 1(1):21–35, 2011.

[11] M. Forsell. Locality-aware Memory System for PRAM Mode Private Data Storage in the CESM Architecture. In *Proceedings of the 18th International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA12), July 16-19, 2012, Las Vegas, USA*, 2012.

[12] M. Forsell and M. Hiivala. Multicore Portability Abstraction. In *Proceedings of the 14th Workshop on Advances in Parallel and Distributed Computational Models (APDCM12), May 21, 2012, Shanghai, China, 772-779*, pages 772–779, 2012.

[13] M. Forsell, P. Hofstee, A. Jerraya, C. Jesshope, U. Vishkin, and J. Traff. HPPC 2009 Panel: Are Many-Core Computer Vendors on Track? In *Lecture Notes in Computer Science 6043*, pages 9–15, 2010.

[14] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of 10th ACM STOC*, pages 114–118. Association for Computing Machinery, New York, 1978.

[15] D. Gajski, D. Kuck, D. Lawrie, and A. Sameh. CEDAR – A Large Scale Multiprocessor. In *Proceedings of International Conference on Parallel Processing*, pages 524–529, 1983.

[16] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley, Reading, 1992.

[17] J. Keller, C. Keler, and J. Trff. *Practical PRAM Programming*. Wiley, New York, 2001.

[18] C. Kessler and H. Seidl. The Fork95 Parallel Programming Language: Design, Implementation, Application. *International Journal on Parallel Programming*, 25(1):17–50, 1997.

[19] V. Leppanen. *Studies on the realization of PRAM, Dissertation 3*. Turku Centre for Computer Science, University of Turku, 1996.

[20] V. Leppanen, M. Forsell, and J-M. Makela. Thick Control Flows: Introduction and Prospects. In *Proceedings of the 2011 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'11), Las Vegas, USA*, pages 540–546, 2011.

[21] G. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8):114–117, 1965.

[22] NVIDIA. NVDIA's Next Generation CUDA Compute Architecture: Kepler GK110, 2012. white paper.

[23] D. Oehmke, N. Binkert, T. Mudge, and S. Reinhardt. Hoe to Fake 1000 Registers. In *Proceedings of the MICRO-38*, 2005.

[24] D. Patterson. The Trouble With Multicore. *IEEE Spectrum*, 47(7):28–32, 2010.

[25] G.F. Pfister, W.C. Brantley, D.A. George, S.L. Harvey, W.J. Kleinfelder, K.P. McAuliffe, E.A. Melton, V.A. Norton, and J. Weiss. The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture. In *Proceedings of International conference on Parallel Processing*, pages 764–771, 1985.

[26] A. Ranade. How to Emulate Shared Memory. *Journal of Computer and System Sciences*, 42:307–326, 1991.

[27] J.T. Schwarz. Ultracomputers. *ACM Transactions on Programming Languages and Systems*, 2(4):484–521, 1980.

[28] A. Seznec. 15mm x 15 mm: The new frontier of parallel computing, 2007. Euro-Par 2007 keynote, August 28-31, Rennes, France.

[29] U. Vishkin. Towards Realizing a PRAM-on-Chip Vision, 2007. Workshop on Highly Parallel Processing on a Chip (HPPC), August 28, Rennes, France (see http://www.hppcworkshop.org/HPPc07/talks.html).